HONEYPHY: A PHYSICS-AWARE CPS HONEYPOT FRAMEWORK

A Thesis
Presented to
The Academic Faculty

By

Samuel Litchfield

In Partial Fulfillment
of the Requirements for the Degree
Masters of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

May 2017

Copyright © Samuel Litchfield 2017

HONEYPHY: A PHYSICS-AWARE CPS HONEYPOT FRAMEWORK

Approved by:

Dr. Beyah, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Meliopoulos
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Owen
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: April 26, 2017

ACKNOWLEDGEMENTS

Many people helped me through my graduate stint at Georgia Tech, and through this work. I would like to thank:

- Dr. Raheem Beyah, my advisor, for helping me through this work and for guiding me through research and academia in general
- Dr. Meliopolous and Dr. Owen for agreeing to sit on my committee
- David Formby, Celine Irvene, Paul Wilson, and Christian Bayens, all graduate school colleagues, for helping me work out the ideas presented here, for helping me proof this document, and for supporting me during this work
- My many friends for supporting me

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Cyber Physical Systems (CPS) are vulnerable systems, and attacks are currently being carried out against them. Some of these attacks have never been seen before, and so the first step in defending CPS is to understand what attackers are doing, and how they are doing it.

Traditionally, honeypots have been a tool used to gain this information, but honeypots need to be convincing to fool attackers. For CPS, being convincing entails not only addressing networking concerns, but also modeling device actuation fingerprints and how the attached process responds to actuations.

In order to create a convincing CPS honeypot, a framework was developed to address the need to present convincing networking, device, and process fingerprints. Two proof of concept systems were developed for this framework, and a set of proof of concept device and process models were implemented.

CHAPTER 1
INTRODUCTION

Since the early 1990s, the idea of entrapping and deceiving computer attackers in order to study their behavior and misdirect them has been used with great success in the computer security field. This practice, traditionally done through a deliberately unused computing system configured to emulate critical resources, called a Honeypot, has revealed many different attacker strategies, allowed researchers to gather malware binaries, and kept more critical computing resources safe. As networking evolved, many honeypots were linked together to form Honeynets, in order to emulate full deployed networks. As attackers learned of honeypots, they improved their techniques to detect whether a resource is being faked, and this is now the primary means of defeating honeynets. If an attacker can realize that a resource is being faked, he can move on to more critical resources, or feed the defender false information in turn.

As the Cyber-Physical Systems (CPS) space grows and becomes increasingly networked, attackers have been more interested in compromising the resources controlling these CPS. Honeypots/Honeynets have been designed to emulate CPS specific components in response. However, all existing CPS honeypots neglect certain aspects of these systems that can alert an attacker to the nature of the honeypot, namely the simulation of the attached physical process and the physics of the devices that interact with the process. Consequently, a new CPS specific Honeypot framework is proposed, called HoneyPhy: A Physics-aware Honeypot Framework, that addresses these problems and aims to be easily extensible to all cyber-physical systems.

1.1 Primary Contributions and Thesis Organization

The primary contributions of this thesis are the following:

- An initial version of HoneyPhy is proposed, which details a general structure for a CPS Honeypot to include process and device models
- A proof of concept is implemented for this initial version of HoneyPhy, modeling a simple HVAC system
- An extended version of HoneyPhy is proposed, which details a specific structure for HoneyPhy deployments, dynamic deployment, and log collection
- A proof of concept is implemented for this extended version of HoneyPhy, modeling a simple water treatment system

The remainder of this thesis is organized as follows: Chapter 2 provides background and related work in assessing Cyber Physical System security and prior attacks, the field of honeypots, and the development of CPS-targeted honeypots. Chapter 3 presents the design of a CPS honeypot capable of modeling process and device behaviors. Chapter 4 presents the implementation of two proof of concepts, and the attendant implementation of the varying process and device models. Finally, Chapter 5 presents the conclusion and potential avenues for future work.

CHAPTER 2
BACKGROUND AND RELATED WORK

2.1 Background: Cyber-Physical Systems & CPS Security

NIST defines a Cyber-Physical System, or CPS, as "co-engineered interacting networks of physical and computational components" [1]. Traditionally, these systems were mostly operated by engineers with a background in controls, and so most of the notions of "security" applied to these systems are defined in terms of operational security. An example of this is the standardization of N-1 contingency analysis by the North American Energy Reliability Corporation (NERC) [2]. As computational technology makes the controlling of these systems easier, however, the increased networking that comes with it increases the potential for insecurity.

The key component that differentiates CPS from traditional computer networks is the attached physical components. These components and the actuation and sensation they are capable of form an alternative means of flow for information between networked components, albeit one that if misused can cause significant physical damage.

2.1.1 CPS-Targeted Attacks

Recently more work has been done investigating the cyber security of CPS. Much of this work has been motivated by the increasing number of CPS-targeted attacks since the publicization of Stuxnet. A set of recent attacks are discussed below.

Stuxnet

Stuxnet [3] is likely the most commonly known CPS-targeted attack. Stuxnet was first noted in 2007, with a later version found in 2010, and was designed to delay or destroy

Iran's uranium enrichment capabilities. It did this by infecting a variety of Siemens controllers. When activated, Stuxnet blinded the human plant controllers by replaying previous signals, and executed control actions that damaged equipment. In the early version, this control action consisted of overpressurizing the uranium enrichment centrifuges, and in the later version, this action consisted of altering the rotational speed of the centrifuge rotors.

Duqu

Duqu [4] was a piece of malware found in the wild, which presented similar fingerprints to Stuxnet. Later analysis by Kaspersky [5] concluded its purpose to similarly be spying on the Iranian nuclear program, although the malware itself did not appear to have the same PLC-targeted approach.

Dragonfly/Havex

Dragonfly [6] is a group that has, since 2013, targeted organizations that deal in the energy sector. The group uses a variety of techniques to compromise company machines. In addition, the ICS software producers that supplied the companies were compromised, and the updates provided by the software producers' websites included malware.

Ukranian Electric Utility

Late December 2015, 3 Ukranian power distribution substations were taken offline by attackers [7]. This attack resulted in 225,000 customers going without power for several hours. The attack used several different methods, but involved developing malicious firmware for serial to ethernet converters, hijacking SCADA clients and HMIs to send commands to breakers, and executing DOS attacks to delay remediation efforts. Evidence indicates the attacks to be highly targeted to the specific utility.

2.1.2 Other Motivating Factors

Other work has highlighted the emerging vulnerability of CPS, as well as attacker interest in them.

Shodan

The Shodan project [8] has gathered a large amount of publicity. It regularly scans a number of ports on the entire IPv4 address space, and logs the banners returned from the relevant services. Many of the ports included in its scans include protocols commonly used by CPS, and consequently one of the major results of the project has been greater insight into the sheer number of CPS devices that face the internet. A common justification for outdated equipment and slow patch cycles in CPS equipment is the assumed 'air gap' between the internet and production networks. Shodan acutely illustrates how often this assumption is violated.

Probing Behavior

Recent work [9] examined the probing and discovery behavior directed towards commonly used CPS protocols by monitoring unused IP space. The work monitors a number of ports for unsolicited traffic and attempts to correlate separate events into orchestrated probing campaigns. Their methodology identified 58 separate probing campaigns, 5 of which are significantly larger, consisting of at least 50 probe sources. Some of the identified probing campaigns originated from benign organizations, but others are inferred as being malicious.

2.2 Background: Honeypots

As stated previously, a honeypot is a computing system designed to be attacked. They have a rich history, dating back to the first instance of using deception to waylay and attacker, divine their intentions, and determine their origin [10].

Traditionally, honeypots are broken into two orthogonal sets of broad classifications: high and low interaction honeypots, and research and production honeypots.

2.2.1 Low Interaction

Low interaction honeypots work to accurately emulate a set of services and some system behaviors. No effort is given to other services, and the emulated services might not implement the services full feature set. Low interaction honeypots get their name from the low level of interaction available between the attacker and the machine. Consequently, low interaction honeypots can behave in unexpected fashions when they encounter unexpected behavior. This unexpected behavior can also alert the attacker to the fact that the machine he is interacting with is a honeypot, which will likely cause the attacker to either disconnect or continue the interaction with the intent to mislead. Additionally, because attackers are limited in how they can interact with the machine, the information that can be gained from the honeypot is also limited. Both of these cases severely limit the usefulness of a low interaction honeypot, and motivated the movement to high interaction honeypots in their place. One very well-known example of a low interaction honeypot is HoneyD [11], which can be configured to emulate a variety of different OS, offering a variety of services.

2.2.2 High Interaction

High interaction honeypots, in contrast, do not emulate. They are real resources, instrumented to log an attackers behavior, and are deployed to be unused for any other purpose. This allows the attacker to interact with real operating systems and applications. Consequently high-interaction honeypots offer great benefits in logging all of an attackers behavior, but also expose great risk in allowing an attacker to potentially compromise these real resources.

High Interaction honeypots have gained popularity since virtualization has become more prevalent. Virtualized environments allow many victim machines to be hosted on

the same physical resource, and the networking conditions to be tightly controlled.

2.2.3 Research

Research honeypots are deployed in order to gain an understanding of the motivations and strategies of attackers. These will, in general, need to be more convincing, as these honeypots need to convince an attacker to maintain access long enough to actually attack the system.

2.2.4 Production

In contrast to research honeypots, production honeypots are deployed within production networks. They primarily serve to notify the network owners of any unauthorized access, which then serves as an early indication of potential compromise. No deep deception is necessary, as any access to the honeypot at all provides value in giving advance warning of an intrusion.

2.2.5 Honeypot Detection Techniques

Many honeypots, but primarily research honeypots, derive their value from how well they convince an attacker of their authenticity. The more convincing a honeypot is, the more likely an attacker is to interact for longer, and to carry out some interesting attack. Consequently, as honeypots became more and more popular, a cat and mouse game emerged between honeypot detection and deception techniques.

Ultimately, detection techniques differ between low-interaction and high-interaction honeypots. Low-interaction honeypots commonly only implement some subset of the services they expose, so detecting a low-interaction honeypot requires testing some corner case of the emulated service, and examining the resulting behavior to see if it matches the real service. An example of this is the detection of the Kippo SSH Honeypot through sending 8 new-line characters and examining the differences between how Kippo responds and

how a real SSH service responds [12]. High-interaction honeypots in contrast are often revealed through more subtle fingerprints. An early high-interaction honeypot tool, called Sebek [13], implemented a kernel-level module to capture keystrokes and other system data. Subsequently, a method to detect and circumvent Sebek was published, called NoSE-BrEaK [14], which detected the presence of the Sebek kernel module in several subtle ways including analyzing how network interface statistics changed and analyzing system call tables.

Ultimately, the requirement for modeling process and device behavior arises from these more subtle methods of honeypot detection seen in high-interaction honeypot evasions. These subtle detection techniques illustrate that attackers are capable of, and willing to, test the machines they interact with in various ways. If an attacker is aware they are interacting with a CPS, and are wary of the possibility of the system being a honeypot, it is possible for them to compare the behavior of the system in relation to the attached process and devices for inconsistencies. This creates a necessity for a convincing CPS honeypot to accurately model these processes and devices.

2.3 Background: CPS Honeypots

Since 2004, a variety of CPS-targeted honeypots have been released and deployed. A listing and analysis of modeling capabilities is provided below.

2.3.1 SCADA HoneyNet Project

The first low interaction CPS targeted honeypot was released in March, 2004 by Cisco Systems, and was called the Supervisory Control and Data Acquisition (SCADA) HoneyNet Project [15]. It leveraged HoneyD [11], Arpd, Snort, and Tripwire to emulate many hosts on a network. Specifically, it aimed at emulating FTP, HTTP, Telnet, and Modbus for a Schneider PLC, and FTP, HTTP, SNMP, and S7comm for a Siemens PLC. The honeypot makes no attempt to simulate process behavior. The project is no longer maintained.

2.3.2 Digital Bond's Honeynet

Released shortly after, Digital Bonds Honeynet [16] has a similar goal of providing a low interaction honeypot simulating a single Modicon Quantum PLC. The system can be configured to either have a virtual machine as a target, with simulated applications and HoneyD monitoring interactions, or be deployed as a high-interaction honeypot with a real device. Both targets are set behind a Honeywall [17] designed to separate the target machine from production networks and filter outgoing traffic.

2.3.3 Dynamic Honeyd System

A system was described in [18], which proposed using several tools to identify other hosts residing in a network, and dynamically configuring and deploying instances of Honeyd. The target environment was a small campus grid located in Idaho Falls, Idaho, and contained many devices traditionally associated with CPS. Significant evaluation was done in the network fingerprints presented by the Honeyd instances, but no mention is made of emulating process or device behaviors.

2.3.4 Gaspot

Gaspot, presented at Blackhat 2015 [19], was based on research done at TrendMicro. Motivated by attacks observed on gas station control devices, the low interaction honeypot simulates basic services provided by these devices, and logs all interactions. The honeypot is relatively simple, and responds to queries with randomized values within plausible ranges. After deployment in a variety of countries, attacker interactions and origins were analyzed.

2.3.5 Honeyd+

In [20], a Honeyd based high-interaction honeypot for a single proxied PLC is presented. Specifically, the system aims to cheaply present many virtualized hosts, all of which are

proxied to the same physical device. It was tested with two different physical PLCs, and two different proxy devices. It is unspecified what other physical components the PLC device is connected to.

2.3.6 Virtual ICS Honeypots-in-a-Box

A honeypot system was presented in [21], based on MiniCPS [22]. The MiniCPS framework is capable of link shaping , as well as Software Defined Network emulation. Similarly to this work, the system provides process and device simulation, but the device simulation consists of emulating device services and logic, and does not extend to actuation fingerprints.

2.3.7 Conpot

Conpot is an actively maintained low interactive server side Industrial Control Systems honeypot designed to be easy to deploy, modify and extend [23]. While inherently extensible, Conpot is not aimed at modeling either processes or devices. System definition is done through xml files, and protocol emulation is done using Python. Out of the box examples simulate device memory, leveraging the Modbus_tk library.

Conpot has been extended to imitate a smart meter in [24]. The imitation smart meter presents Modbus, SNMP, and a static HTTP HMI. Because the honeypot is primarily providing intelligence to a production system, the focus is on using any interaction as an indication of compromise. Consequently, little work is done towards process or device emulation.

Conpot has also been extended to simulate electrical grid components in the project GridPot [25] GridPot leverages GridLAB-D to model an IEEE power distribution test case, and this is used as a process model.

2.3.8 CryPLH

CryPLH, the Crysys PLC Honeypot [26] is an actively developed low-interaction honeypot designed to emulate a Siemens Simatic 300 PLC. It simulates the exposed HTTP, HTTPS, SNMP, and Siemens SIMATIC STEP7 (carried out over the ISOTSAP protocol) configuration interfaces on a minimal Ubuntu Linux VM, and uses a central configuration file to simplify and end users configuration. The HTTP/S and ISOTSAP interfaces both have logins where no username/password combination will successfully log in, and the visible web portal does not change to reflect the PLCs environment.

2.4 Why Existing CPS Honeypots Are Not Sufficient

In traditional network focused honeypots, as well as in existing CPS honeypots, the main goal was to emulate the kinds of protocol quirks that fingerprinting utilities like Nmap and p0f look for. However, CPS honeypots should provide auxiliary information arising from the attached physical system. This auxiliary information is both the ability to compare the moment to moment state of the CPS for consistency (i.e., leveraging the physics of the process and sensors), as well as observing the individual connected devices for unreasonable actuation fingerprints. If either the process physics or device actuation times are unrealistic, an attacker can easily determine if they are in a honeypot.

A simple example can illustrate why process and device simulation are important to the design of a CPS honeypot. A consumer home Heating Ventilation and Air Conditioning (HVAC) system represents a familiar and intuitive CPS, where networked thermostats control physical devices like heaters, compressors, and fans. In reality, if a command is issued by a thermostat to begin heating, a heater turns on. If temperatures are read in succession, the home temperature can be seen to slowly rise in response. Imagine that an attacker is interacting with a honeypot designed to emulate this system. First, the attacker turns on the heater, and then he closely monitors the homes temperature sensor. If this honeypot makes

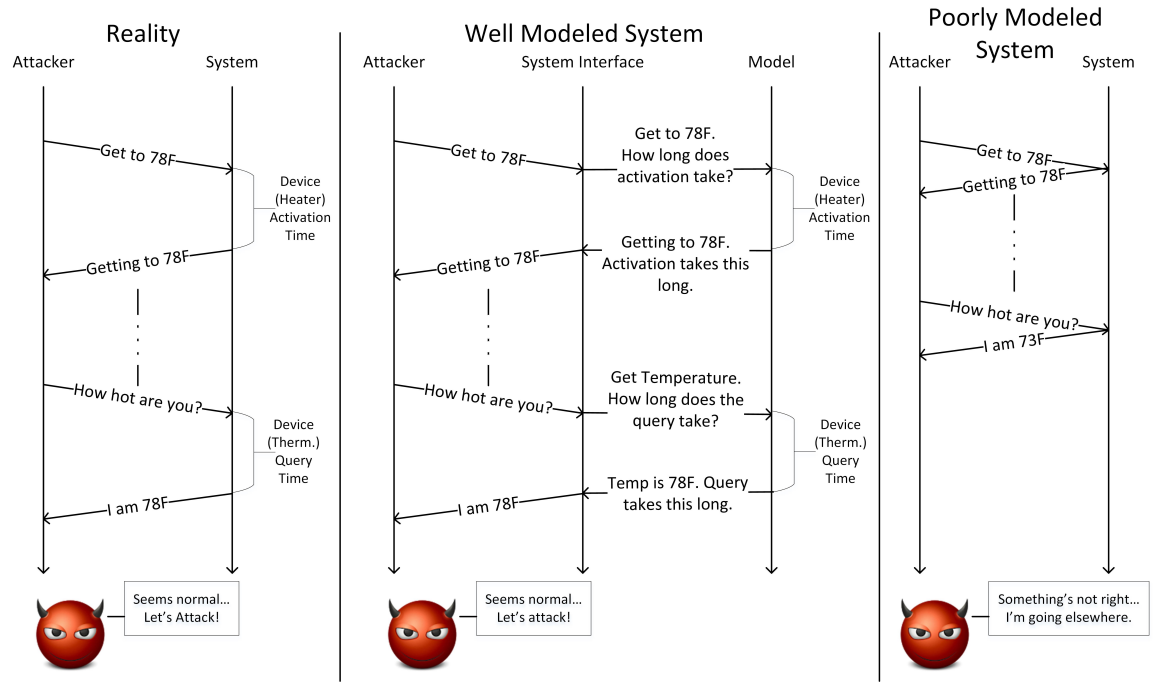

Figure 2.1: Outcomes of an attacker interacting with different levels of simulation

no attempt to simulate the process it claims to control, and instead returns random responses or does not respond to the heating, an attacker will see temperatures over time that do not reflect the activation of the heater. Alternately, the temperature sensor could instantly show the final temperature, which would completely neglect the physics of the system. In either scenario, the attacker knows the system he is interacting with is either faulty, or does not control the system it claims to. Accordingly, they are likely to not continue interacting with the system, and the honeypot loses utility.

A similar example can illustrate the need to simulate the physical/mechanical fingerprint of a device (i.e., actuator). If the thermostat controls a heater through the use of a mechanical relay, activation of the heater requires changing the state of that relay. This state change requires some amount of time, determined by the electro-mechanical characteristics of the relay [27]. In some devices, this delay can be on the order of milliseconds or longer, and the distribution of this delay forms a fingerprint for a device. If, as is the case in many other kinds of CPS, the thermostat is instrumented to confirm the state change, this

delay is now exposed to the attacker. An attacker can look for this device delay, and use it to test whether the system is a honeypot. This is illustrated in Figure 2.1. The left scenario illustrates an attacker interacting with a real system, so all delays and responses result from physical process and device behavior. The right scenario illustrates an attacker interacting with a CPS honeypot that does not attempt to model process behaviors and device delay, and the lack of this delay and deviations from expected process behavior will alert the attacker to the honeypot. The middle scenario illustrates an attacker interacting with a CPS honeypot that is capable of modeling both process behavior and device fingerprints. Responses provided to the attacker are thus realistic, and the attacker proceeds to perform observed malicious actions.

These scenarios present an actuation device whose fingerprint is a simple delay characterized some distribution, but devices could conceivably have much more complicated fingerprints. There exist mixing valves that receive an open/close set point from a PLC, and can return a signal to the PLC showing the current state. The response of the device to changing the set point can be used to fingerprint the device.

Table 2.1 shows the state of emulation provided by the previously noted CPS honeypots. Most of them neglect modeling process behavior, and all neglect to model device actuation times and fingerprints. In the table, the term proven either indicates an attacker was fooled by the honeypot at this level in the honeypots history, or that the honeypot was tested by an associated tool by the honeypots creator.

2.4.1 How Should a CPS Honeypot Fit Into Honeypot Classifications?

Pure high-interaction honeypots are fundamentally unsuited to CPS, because they rely on either deploying another physical copy of the resource in question, or somehow virtualizing it. Deploying a copy of an entire CPS with the express purpose of being compromised exposes the same safety risks as the original system, and imposes large costs. A pure high interaction honeypot can be deployed for a single component or small group of components

Table 2.1: Modeling Capabilities of Existing CPS Honeypots

	Human Interaction	Network Stack Emulation	System & Process Simulation	Device Modeling
SCADA HoneyNet [15]	Proven	Honeyd provided	None	None
DigitalBond Honeynet [16]	Proven	High-Interaction Capable	None	None
Dynamic Honeyd System [18]	Unmentioned	Honeyd provided	None	None
GasPot [19]	Proven	Unmentioned	None	None
Honeyd+ [20]	Proven	Proven	None	None
Virtual ICS Honeypots-in-a-Box [21]	Proven	Focus on network emulation	Modeled in Python	Confined to computation logic
Conpot [23]	Proven	Proven through testing	None, but extensible [25]	None
CryPLH [26]	Proven	Nmap discernable differences	None	None

within a CPS, but without the physical portion of the system to interact with, the usefulness of these honeypots is limited.

One solution is to create a hybrid-interaction honeypot, where real devices (e.g., programmable logic controllers (PLCs), intelligent electronic devices (IEDs), and remote terminal units (RTUs)) and interfaces interact with process and device simulations that can effectively fully replicate the behavior of the CPS process. In some cases obtaining and linking real devices might be impractical, however, such as if honeypot instances need to be dynamically deployed. In these cases, support must be made available for virtualizing real devices where possible, or creating device models emulating real functionality where necessary.

For this thesis, obtaining real devices was impractical, so the implementations developed here are entirely low-interaction. This is a result of the device models used, however, and the framework developed aims to ease the integration of a wide variety of models.

CHAPTER 3
HONEYPHY DESIGN

In order to address the limitations of current CPS honeypots, a new type of CPS honeypot is proposed which can account for the physics of the process and devices to which computing systems are attached. To begin with, an initial set of requirements are defined.

3.1 Initial Requirements

This new CPS honeypot should correctly model software and protocol fingerprints, as with all other honeypots. Matching these fingerprints is well understood, and the goal of all previous CPS honeypots. This interface layer of the honeypot could be either high or low interaction, depending on access to equipment such as Human-Machine Interfaces (HMIs).

In addition to this, this CPS honeypot should correctly model the behavior of the physical system. This primarily ensures that physical parameters, when queried, behave in a way consistent with attacker expectations. Without this, future attackers could conduct simple tests to check the authenticity of the machine or machines they are interacting with. A CPS process model will require simulation to model this behavior, as replicating the process exposes real risks.

Finally, this honeypot should correctly model the fingerprints introduced by the constituent devices within the CPS. These fingerprints could either originate from the operation of real devices used in the CPS honeypot, or be generated by modeling the devices. Returning to the HVAC example, these fingerprints would originate from the time it takes for the electromechanical relays to physically open or close, energizing or de-energizing the fan, heater, or compressor to move the system to the desired state. Attackers measuring the physical actuation time for these devices could detect a honeypot environment if the measurements lay outside the known operation times for each device. These operation

times can be modeled, and these models can be generated in one of two ways, white box modeling or black box modeling.

Black box modeling is the method of generating a model for the behavior of the device based on physical access to it and a set of true measurements of its behavior. By comparison, white box modeling requires no physical access to the device and involves mathematical modeling of the device based on estimates of the device parameters and standard physical models. The primary advantage black box modeling holds over white box modeling is that it results in the most accurate representation of the device behavior, since it is based on empirical measurements. However, attackers will not always have physical access to a target device type to make empirical measurements on to form the black box model. In this scenario intelligent attackers can then resort to white box modeling to generate an estimate of the device behavior and still make educated guesses about whether the device is a honeypot or a true target. Previous work [27] shows that not only is the construction of these models feasible, the models are convincing.

Both these initial requirements and the extended requirements that follow are summarized in Table 3.1.

3.2 HoneyPhy in the Abstract

In order to satisfy these requirements, an initial abstract framework was developed, which here is referred to as HoneyPhy in the Abstract. This framework is composed of three major components: the Internet Interface(s) Module, the Process Model(s) Module, and the Device Model(s) Module. A framework overview can be seen in Figure 3.1. This abstract framework was created in order to provide a means to identify the key components of a process, identify how those components map to models, and provide a common language to configure and interconnect those models. A proof of concept implementation is described later.

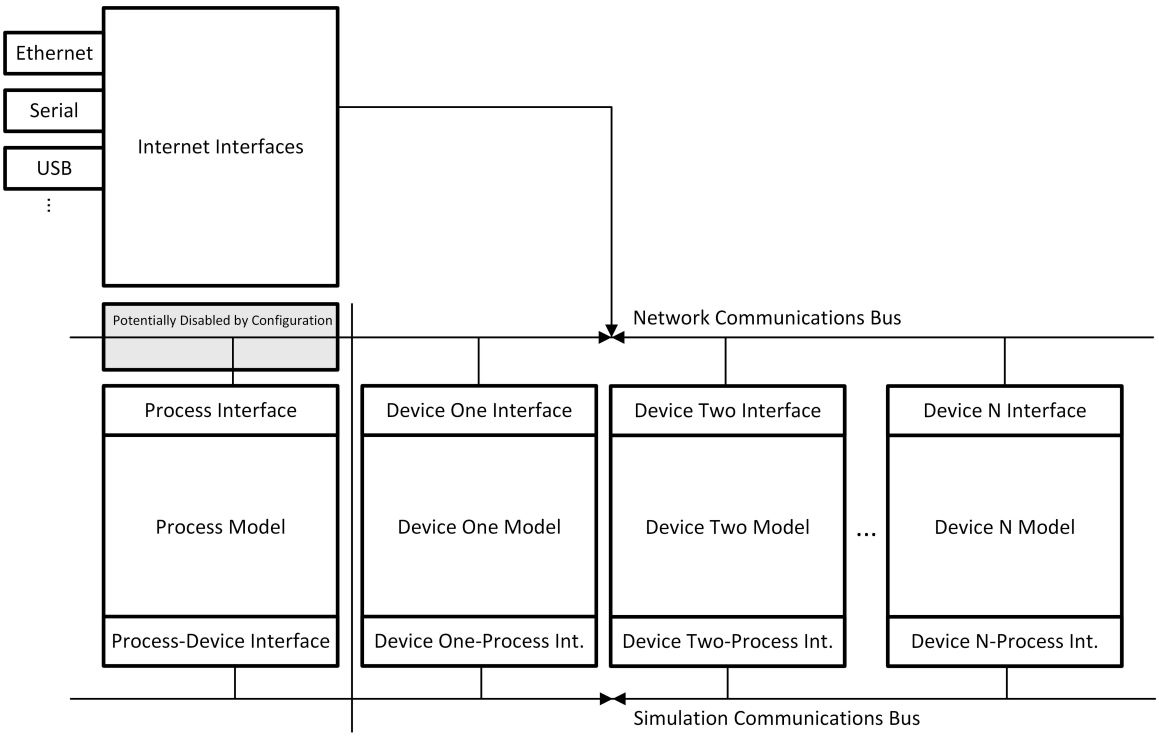

Figure 3.1: Abstract HoneyPhy Architecture

3.2.1 Internet Interface(s) Module

The Internet Interface Module exposes the declared interfaces at the declared addresses. It maintains connections and multiplexes them to their destinations while ensuring outgoing packets reflect the network fingerprint for each device, modifying them if necessary.

3.2.2 Process Model(s) Module

The Process Model exists to simulate the physical process in question. This is essentially whatever is controlled by the devices, and examples include the flow of electricity through the grid, the flow of water through pipes, and the way in which a space heats in an HVAC system. It is interrogated and acted upon by the other devices modeling sensors and actuators, and simulates the process in real time. The Process Model communicates with the various Devices and Models over a separate databus. If desired, a secure Internet-facing interface can be opened to remotely interact with the process model directly. Process Models

could consist of LabView simulations, replays of empirically observed responses (as done in our proof-of-concept presented later), or more traditional controls system models such as a Linear Dynamical System or Auto-Regressive models [28, 29].

3.2.3 Device Model(s) Module

The Device Models encompass all devices found within a CPS, from PLCs to relays to pumps. Individual device models likely map to individual, physically discrete devices. Wherever they control actuators, device models are capable of changing a process variable, and wherever they control sensors, device models are capable of querying a process variable. Where device models simulate computing devices such as PLCs, the models implement logic to simulate the programs those devices run. This can include interpreting incoming queries and responding with the value they obtain by querying the process model, or executing incoming commands by modifying the process model. Where Device Models simulate mechanical devices such as relays or valves, the model implements the developed fingerprint for that device in whatever way is necessary to accurately model the device's behavior. Device models can range from very simple low level black box timing models to real devices, sufficiently instrumented to interact with the process model.

3.2.4 Inter-module Communication

While this abstract framework does not specify a required inter-module communication method, it does specify where communication must be available. The Internet Interface module must be able to route incoming/outgoing communications to all applicable devices, and optionally expose the internet facing interface for the process model. Additionally, all devices must be able to talk not only to each other, but also to the process model. This necessitates a separate Process/Device databus.

3.3 Extended Requirements

Once the initial abstract framework was developed, a more clear idea of the requirements for a deployable version of HoneyPhy was obtained. Significant inspiration for these requirements was derived from modern dynamic and distributed honeypot efforts such as [30, 31]. These requirements fall into several basic categories.

Both these extended requirements and the initial requirements they refine are summarized in Table 3.1.

3.3.1 Support for Physical Devices

For the most realistic honeypots, device models would consist of the devices themselves, sufficiently instrumented to interact with all other models. This would fully replicate all device behavior such as potential exploits and network timings, and effectively create high-interaction honeypots. However, the need to configure and manage physical devices hampers the potential for automation deployment, as discussed in the next section.

3.3.2 Automated Deployment

It is common among more recent honeypot offerings to dynamically deploy a large number of 'sensors' in a network [30, 18]. For implementations of HoneyPhy which are entirely virtualized, there should be a mechanism to support this automated deployment of various sensors. To that end, there should exist a central server that contains all the information necessary to deploy a HoneyPhy instance. This central server should also be capable of being a central repository for the log information generated by the various deployments. As discussed in Chapter 4, in the proof of concept this takes the form of an HTTP server, exposing a REST API.

For implementations of HoneyPhy which are not entirely virtualized, and instead contain virtual models combined with physical components, this same central repository should

be able to be run locally, and should still serve as a log collection mechanism.

As a consequence of the desire for dynamic deployability, it is necessary to separate what is termed 'template' information from 'deployment' information. Template information consists of the code that implements process and device models, as well as services exposed to the internet. In comparison, Deployment information consists of the information specific to an individual deployment, such as MAC addresses, IP addresses, and identifying information found in devices.

3.3.3 Augmented Intelligence Extraction

Historically, honeypots have generated two classes of information based on whether they are research or production honeypots. Research honeypots yield information related to attacker methods and identities, whereas production honeypots primarily yield information related to the presence of attackers. This information arises solely from the network interactions observed.

With the addition of the ability for attackers to interact with processes and devices, it becomes possible to label attackers in a more meaningful way in both scenarios.

Research Honeypots

For research honeypots, the addition of process models and behaviors allows investigations into not only how attackers take advantage of network vulnerabilities, but also how attackers exploit those vulnerabilities to affect changes in process behaviors. Furthermore, the ways in which attackers manipulate process models could give insights into attacker motivations.

Production Honeypots

Similarly, for production honeypots, a reasonably simple implementation attached to a process model could be used to not only give information on the presence of attackers, but

also on the level of sophistication of those attackers. Previous work [32] has shown the feasibility of classifying attacker sophistication based on observed exploits. For production CPS, attacker familiarity with the production system and CPS in general is of similar importance. Network and process logs can be analyzed to determine if an attacker is unaware of the CPS nature of the network, if an attacker is aware of the network's nature but unfamiliar with the devices deployed, or if an attacker is familiar with the deployed system and their first action is to exploit and actuate a device.

Required Augmentation

In order to support providing this increased level of information in both a research and production environment, it is necessary to keep a complete log of process behavior, device interactions, and network communication across all devices. Additionally, safety constraints for process behaviors should be noted, so that obviously dangerous behavior can be easily identified.

All of these various logs should be collected and centrally located. In order to ease post-processing and alert generation, all logs should be timestamped from a single source.

Optional Augmentations

This system could, in addition to the information described above, offer insight into how attackers find vulnerable CPS. The system could check whether or not IPs exposed by the deployment are listed in locations that commonly list vulnerable systems. Shodan [8] is a public example, but other locations could also be used. Frequencies of attacker interactions before and after discovery in the various locations could be compared.

Insights could also be derived concerning how attackers find exploits. CPS exploits, as with traditional exploits, are commonly published. It is also common for these exploits to be collected by a single organization. For CPS related vulnerabilities, this organization is the U.S. Department of Homeland Security's ICS-CERT [33]. Signatures of these pub-

lished vulnerabilities could be derived and compared against intrusion actions. Frequency of usage of the published vulnerabilities before and after publication could be compared. This is obviously less effective for zero-day exploits, but long-term deployment and accurate log-keeping could allow retroactively published vulnerability signatures to be compared to previous intrusions.

For virtualized device models which present network services to be convincing, as many published exploits will be emulated within the device models as possible. Previous honeypots such as Nepenthes [34] and its' successor Dionaea [35] are examples of previous low-interaction honeypots that have emulated exploits.

Augmented Intelligence Example

An example illustrating the way in which this intelligence could be derived is shown in Figure 3.2. Prior to an interaction, the deployment's presence in vulnerable listings and published vulnerabilities effecting the deployment's devices are noted (0). At some point before the intrusion, the attacker finds the vulnerable device and an exploit for that device (1). The order in which the attacker finds this information is irrelevant. The attacker then begins interacting with the vulnerable device, which appears in a realistic place but all traffic is routed to wherever the hardware is physically located (2). All interactions are logged, but if the attacker takes some action which actuates on or senses process state, the relevant device models exchange the necessary communications, and process state changes to reflect the actuation/sensation (3). When the interaction is finished, all logs are collected centrally (4). These logs can be analyzed to determine what devices the attacker thinks they are interacting with, as well as the attackers familiarity with the specific system (5).

3.3.4 Ease of Implementation

Even small scale CPS can be very complex, and include complex models. A major barrier to deployment for a honeypot that requires complex system models is the amount of effort

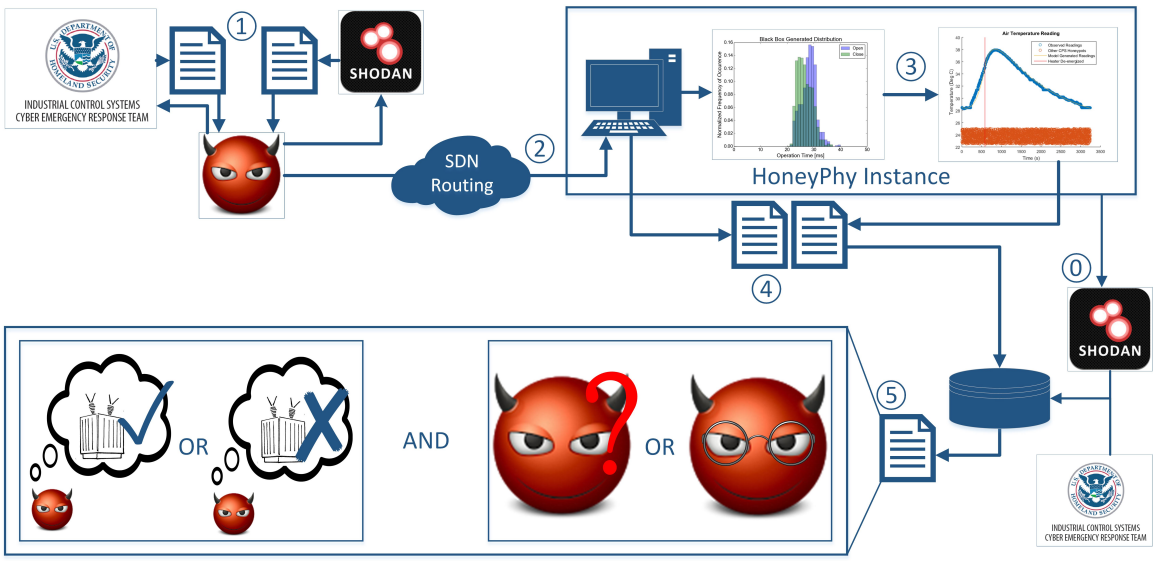

Figure 3.2: Additional Honeypot Intelligence Example

needed to implement those models, and integrate them into the honeypot. Similarly, a primary identifier for honeypots like Conpot on systems like Shodan are default, unchanged configuration parameters.

Any deployable HoneyPhy system should make it as easy as possible to integrate previously developed models and services. This could take a number of forms, but ultimately the system should minimize the amount of code a user should have to write to implement their desired system. Similarly, the system should also have as simple a configuration process as possible to minimize the possibility of users leaving default parameters.

3.3.5 Realistic IP Address Placement

Finally, in order to appear convincing, the IP Addresses presented by the various devices of a CPS must be realistic. In this context, 'realistic' corresponds to the IP addresses being allocated to a single company that could believably control the modeled system and that operates in an area corresponding to the geo-location obtainable for those IP addresses, such as the information obtainable through Shodan [8]. In [36], a method is presented to achieve this through wormholes.

Table 3.1: Summary of Requirements For CPS Honeypot Design

Initial Requirements	
Networking Fingerprints	Honeypot should correctly model protocol fingerprints
Process Behavior	Honeypot should correctly model process behavior
Device Fingerprints	Honeypot should model device actuation fingerprints
Extended Requirements	
Physical Device Support	Design should allow for incorporation of actual devices
Automated Deployment	Design should allow for multiple instances to be deployed dynamically
Augmented Intelligence Extraction	Design should allow collection of all information from the deployment, including process and device logs
Ease of Implementation	Process and Device models should be simple to integrate with each other. External process models should be supported
Realistic IP Addresses	IP Addresses should belong to the organization that owns the modeled system

Ideally, realistic IP addresses would either be provided by a system owner for a deployment in a research setting, or naturally be available to the system owner in a production setting. As the system implemented and presented here was not created with collaboration with any system owner, obtaining realistic IP addresses was not possible.

3.4 Deployable HoneyPhy Structure

In order to satisfy these refined requirements, another iteration of the framework was created. This updated framework is referred to as the Deployable HoneyPhy. An overview of this system can be seen in Figure 3.3. Individual deployments resemble the abstract framework, and device and process models map in the same way. This is shown in Figure 3.3 using models implemented in the abstract framework proof of concept, discussed in Chapter 4. Similarly, there is an external network where devices expose services and an internal network where devices communicate actuations and process sensations.

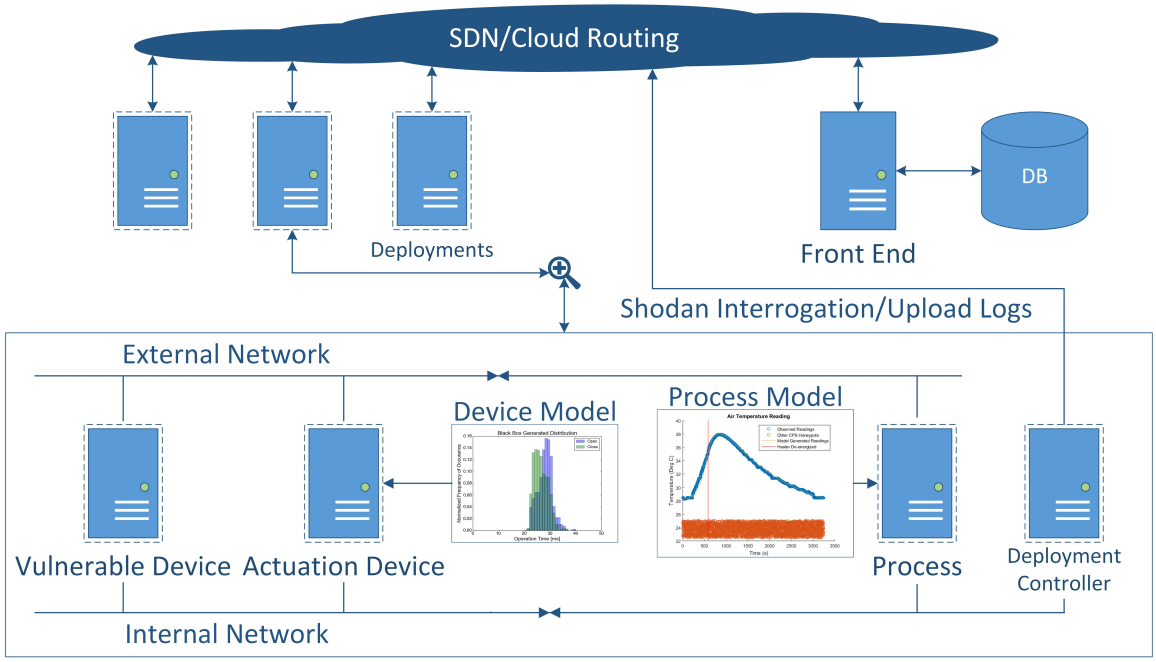

Figure 3.3: Concrete HoneyPhy Architecture

In contrast to the abstract framework, there exists a front end and central repository which contains all the necessary template and deployment information, backed by a database. It also acts as the central point for the uploading of process, network, and device logs.

There is additionally a deployment controller which performs several functions. On startup, it queries the front end to obtain the template and deployment information for its specific system. The deployment controller is optionally responsible for checking for the presence of deployment IPs in locations known to identify vulnerable CPS devices. Finally, the deployment controller is responsible for collecting process, network, and device logs, and uploading the collected logs to the front end.

CHAPTER 4
HONEYPHY IMPLEMENTATION

In order to prove the feasibility of both versions of HoneyPhy, two proof of concepts were created.

4.1 Abstract HoneyPhy Proof of concept

In order to prove the feasibility of the Abstract HoneyPhy, a proof of concept was constructed, modeling a simple heating system. The process and devices included were modeled.

4.1.1 Physical System Architecture

The simple heating system consists of an insulated and heated volume, a set of components acting as a thermostat, and a set of components acting as a heater. The logical architecture of this system is presented in Figure 4.1.

The thermostat analogue labeled in Figure 4.1 consists partially of a personal computer executing temperature control logic based on the temperature read from a USB thermometer located inside the heated area. If this temperature violates the programmed limit set points, the PC sends a DNP3 command, using the OpenDNP3 library, to a connected SEL-751A Relay to either turn the heater on or off. The SEL-751A Relay responds to that command by closing or opening a physical relay (Potter and Brumfield (P&B) KUL-11D15D-24), for which a black box model was previously obtained.

This physical relay, along with the ceramic heater bulb it controlled, formed our heater analogue, as labeled in Figure 4.1.

While not implemented in the system, in a real Internet of Things (IoT) thermostat the current temperature, heater status, and set points could be interrogated and controlled

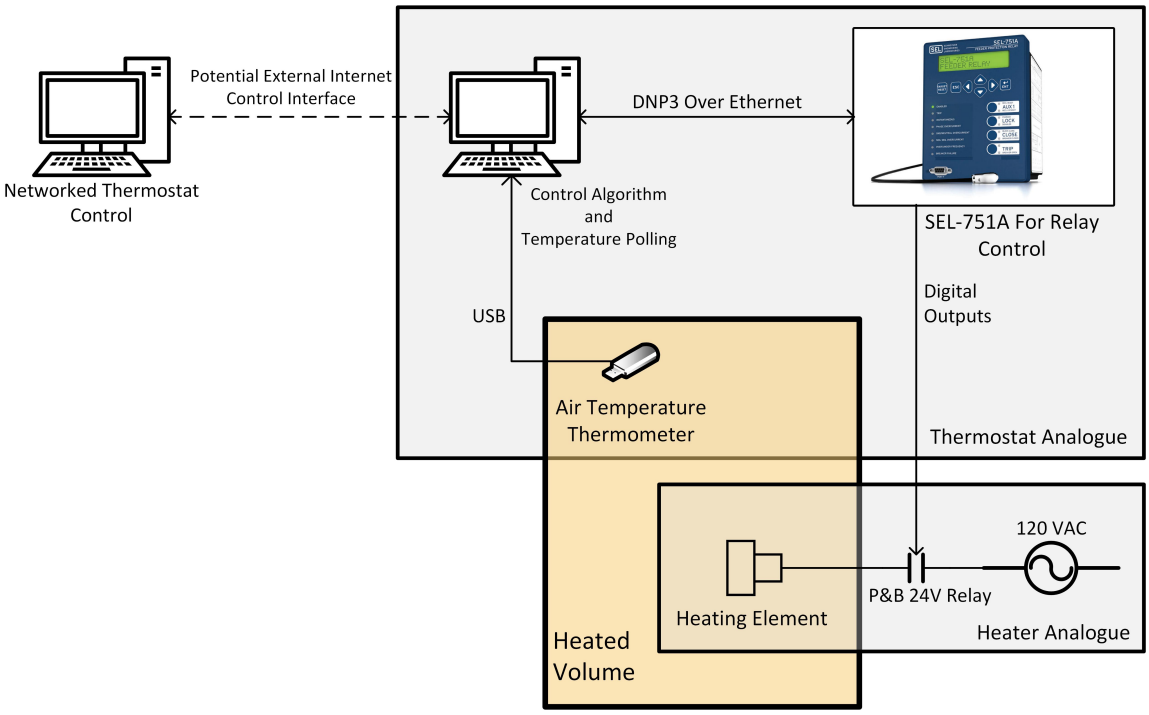

Figure 4.1: Process Modeled in Abstract Proof of Concept

through an external network interface of the PC.

4.1.2 Models and Simulation

In order to simulate this system, an analytic model was developed of how the enclosed volume was heated by the heating element and was cooled by the environment. This was based on empirical results gathered from the internal USB thermometer, and made up of closed form equations, seen in Equations 4.1 and 4.2, where T_bulb , T_air and T_env represent the temperatures in degrees Celsius of the bulb, air, and environment respectively, t indicates time, and S_heater represents the state of the heater. As the model looped through the simulation, the time between the last state update and the current state update are used to calculate the update in observed temperature for both the internal heater and the air temperature. The specific coefficients in the equations resulted from creating the general form of the equations from Newtons Law of Cooling, then fitting the resulting equations to the observed data. This analytic model was used for the process model portion of the

simulation.

$$BulbTemperature(^{\circ}C) = BulbTemperature + \Delta t \cdot [(HeaterState) \cdot (0.00775 \cdot BulbTemperature^{1.04}) - 0.00084 \cdot (BulbTemperature - AirTemperature)^{1.48775}] \quad (4.1)$$

$$AirTemperature(^{\circ}C) = AirTemperature + \Delta t \cdot [0.00084 \cdot (BulbTemperature - AirTemperature)^{1.085} - 0.00041 \cdot (AirTemperature - EnvironmentTemperature)^{1.225}] \quad (4.2)$$

The top of Figure 4.2 shows an observed heating/cooling curve from the physical system, and a similar curve generated by the process model for the same control actions. This curve occurs above room temperature, energizing the heater at the beginning of the data set at 28.5 C and turning it off at the vertical read line. Assuming these control commands originate from the attacker, if they energize the heater at t = 0s, at t = 500s they can compare the reported temperature of the system against either their intuition of the system or a separate process model they have created. Data points have been superimposed below the real data to illustrate what Digital Bonds SCADA Honeynet and Gaspot would return when asked for the air temperature, if used to simulate a similar system. This data was generated by inspecting relevant source, and observing the use of Python's random.randint for Gaspot, and finding only classes such as RandomDigitalIn and RandRangeInputRegister when inspecting the Modbus simulator for Digital Bond's SCADA Honeynet. This clearly does not capture process behavior, and when the attacker checks the reported behavior of the system at t = 500s, it will be clear that something is wrong.

To capture device behavior, previously developed black box models for the same relay

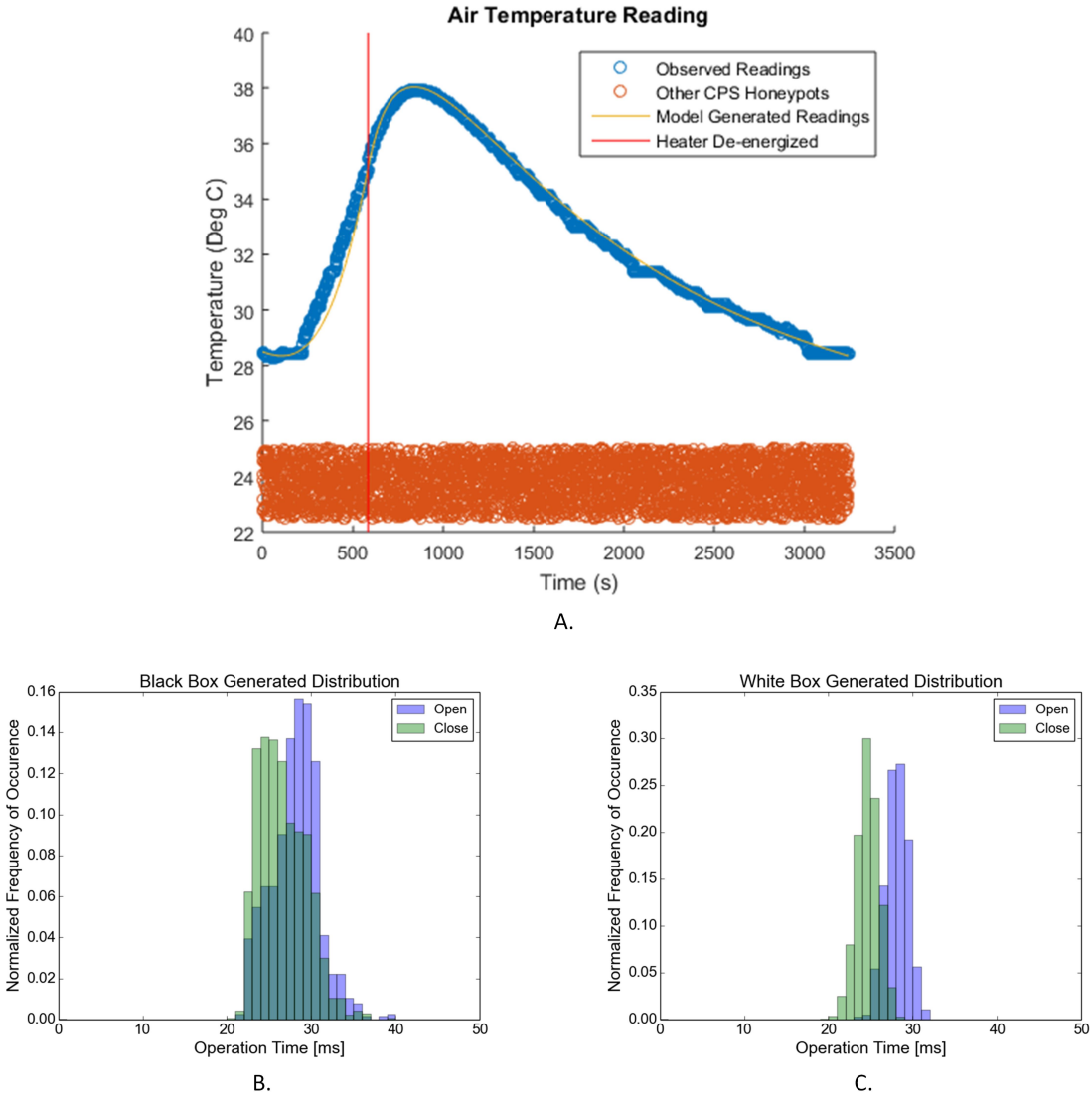

Figure 4.2: Models Used in Abstract Proof of Concept

were leveraged [27]. For the implemented device model, the black box data was randomly sampled. A white box model was also previously developed for the same relay. The simulation is convincing with both models, as can be seen in the bottom of Figure 4.2, which shows histograms of device operation times from both models. Both open and close distributions for both models are shown, and dark regions show overlaps in the open and close distributions. The black box model was generated by empirically observing the device, while the white box was generated by simulating mechanical properties based on specifications. While the standard deviation of both distributions differs slightly due to simplifications made in the white box modeling, the means of both models are clearly similar. The present differences should not be apparent to any shorter term attackers. Machine learning tools in [27], trained using this white box model, successfully differentiated two models of physical relays with 80% accuracy.

The modules used in implementing this proof of concept were all multithreaded and written in Python, using standard system libraries. The system is capable of real-time simulation, at a variety of time-step granularities.

4.2 Deployable Proof of Concept

In order to move HoneyPhy towards a system that researchers and production networks can actually use, a proof of concept implementation for the Deployable HoneyPhy framework was implemented.

4.2.1 Framework Elements Common to All Systems

The following portions of the framework were implemented in the proof of concept, and are common to any system modeled within the framework.

REST Framework Implementation

A simple centralized template configuration, deployment configuration, and log collection repository was created using the Django REST framework [37, 38] in Python 2.7. This framework provided the means to store related configurations backed by a SQLite 3 database. With this, it was possible to only provide deployment controllers with the id of a desired deployment, and the deployment controllers are capable of retrieving all related configurations, and upload log collections tagged with the related deployment. The implemented REST framework also optionally serves deployment controller source, and the deployment controller is capable of polling the REST framework for ready deployments. During testing, only a single bash script was required to retrieve and start a deployment.

While a web-based front end that enables remote inspection of logs would be the ultimate goal, for the proof of concept Django's administration framework was sufficient to manage deployment configurations, template configurations, and log collections. A screenshot of the administration page for editing a deployment configuration record can be seen in Figure 4.3.

General Implemented Deployment Structure

An overview of how the deployable HoneyPhy proof of concept was implemented can be seen in Figure 4.4. All device and process models run on the same physical hardware, but run within docker containers. Every device has a central device controller. For every service a device implements, a separate thread is run, which is connected to the central device controller via Python Multiprocessing Pipes. The process model in this framework is just a separate device, with a service that implements the math necessary to model process evolution. Devices communicate among themselves over UDP through an internal network, which is implemented as a Docker bridge network. These device services are optionally connected to an external network, which is also implemented as a Docker bridge network and exposed through the physical network interfaces of the deployment machine. A con-

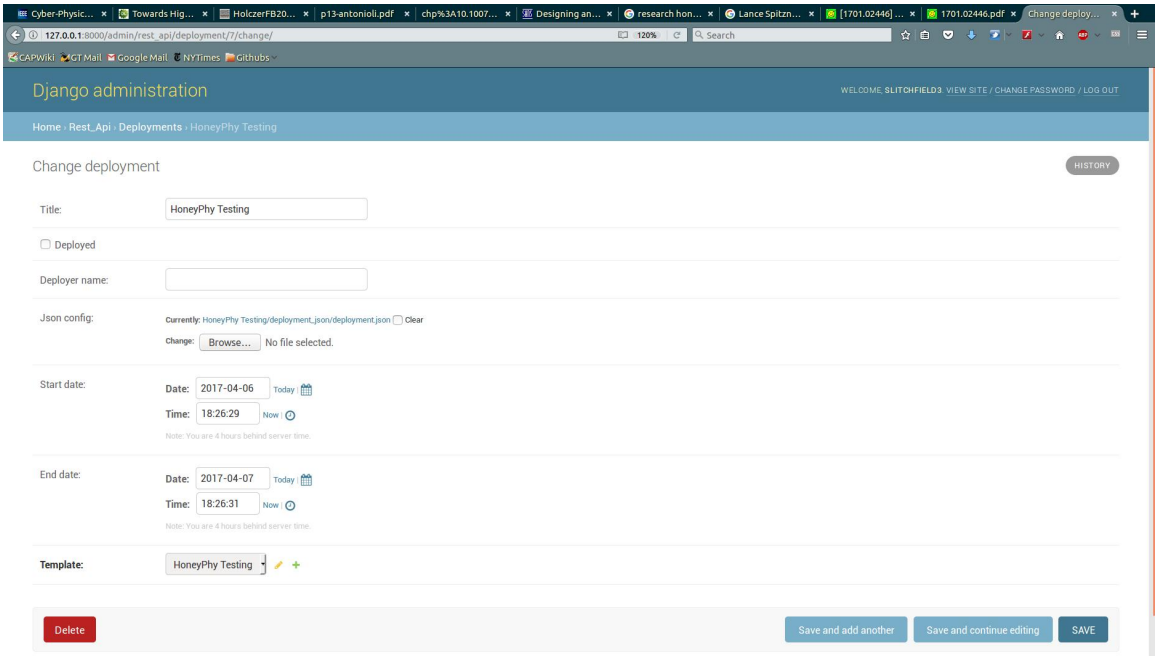

Figure 4.3: Screenshot of Django Administration Site

tainer running Snort captures traffic from this external network to get a complete record of the network interactions that occur within the system.

More detail information on the individual components identified here is given below.

Deployment Controller Implementation

The deployment controller for the deployable HoneyPhy framework was implemented in Python 2.7. Upon startup, the system polls the REST framework for deployments that have not been deployed yet, and claims the deployment with the earliest activation time. Once it receives confirmation that the deployment is claimed, it downloads the necessary configuration zip files, and begins readying for activation.

Once the activation time of the deployment passes, the deployment controller begins spinning up the deployment. In this framework, each device is implemented in a Docker Container [39], which, when combined with the Python Docker API [40] eases deployment. Each device shares the same base docker image, and device specific configuration and services are mounted as a volume. Additionally, network monitoring and alert generation

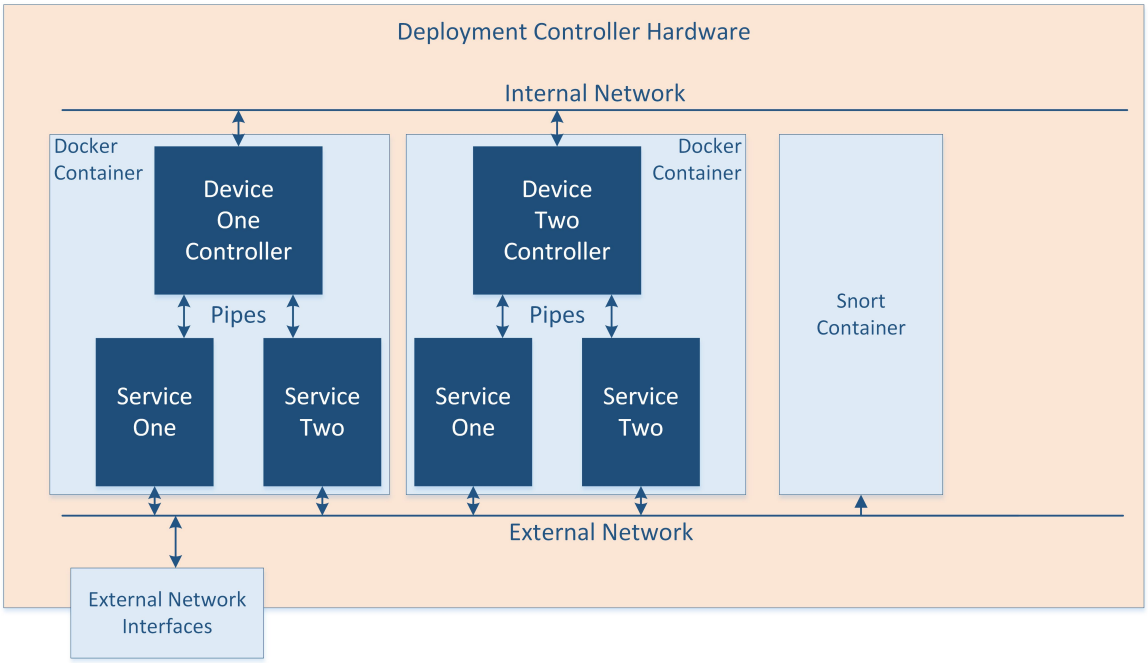

Figure 4.4: Overview of the Deployment System

is accomplished through the use of Snort [41], which is also run within a docker container. The first thing the deployment controller does, then, is build the device docker image and the Snort docker image and container.

The deployment controller creates the internal and external networks as Docker Bridge networks. In [42] it is shown that Docker bridge networks are slower than native bare-metal networking, but should still be insignificant in comparison to other networking delays.

The deployment controller then parses the template configuration file and deployment configuration file, and creates individual device docker containers. Template configuration files will always specify an interface a device uses to communicate over the internal network, so devices are connected to the internal network when created. Deployment configuration files can possibly specify an interface devices host a service on, and if so, the device is connected to the external network as well. Device configuration is passed as an environment variable to the Docker container.

The deployment controller then actually starts the devices and the Snort container. At a tunable interval, the deployment controller gathers all the generated logs, compresses them,

and uploads the collection to the rest framework. In the same interval, the deployment controller polls Shodan [8] for the information Shodan has collected on the devices in the deployment. If the controller notices that the current time is after the scheduled end of the deployment, the controller terminates all devices and uploads a final compressed log collection.

Device Controller Implementation

The device controller implements much of the functionality common across any device model. It is passed configuration information through an environment variable in its host Docker container, and any necessary static files are mounted as a volume in the container.

Initially, the device controller parses its configuration information into local data structures.

These first of these data structures specifies information related to the process variables the controlled device 'originates' or requires. For example, a device model implementing a valve 'originates' the status of the valve, and a device model which implements process evolution likely requires the status of the valve. Additionally, devices note which variables their services are dependent on, and which remote devices originate those variables.

Another data structure is the list of services the controlled device implements. The device configuration includes service names, python modules that implement those services, and interfaces those services expose to the external network. Once the data structure is filled, a Python Multiprocessing process is created for each service.

Once all preparation is complete, all the service processes are started, and the device controller begins processing communications originating both on the docker container's internal network interface, as well as the various pipes connected to the device's services. Some of these communications involve the device controller sampling specified process variables from remote devices.

Generic Service Implementation

In order to agree with how the device controller initializes and starts services, there are several requirements for all implemented services. They must be implemented in a class called Service, which implements the method `start service`. A `generic service` class is provided which other services can inherit from, which implements common functionality for other services.

Deployment Timeline Example

In order to more clearly illustrate how the deployment controller activates a deployment, and example is provided in Figure 4.5. First, the deployment controller gets a list of systems ready to be deployed from the REST framework (1), and downloads the configuration files for one of the systems (2). Then, when the deployment's activation time is reached, the controller creates the device container image and Snort container image (3). The deployment controller creates the internal and external networks (4), and finally creates and starts all the devices specified in the system's configuration (5). Once all necessary components are started and the system begins its' simulation, logs are periodically collected and sent to the central server (6).

Network Traffic Capture

In order to capture all network traffic that occurs over the external network, and attempt to identify intrusions by their network fingerprint alone, the Snort system is run within a docker container in two of its modes. First, Snort is run in its packet logging mode to capture all observable traffic. This process is started and killed over a tunable interval, but for the proof of concept this was around 5 minutes. There are interesting problems surrounding the tuning of this capture interval, including intrusions that span multiple capture intervals and the delay between intrusion and alert generation, but they are beyond the scope of this proof of concept.

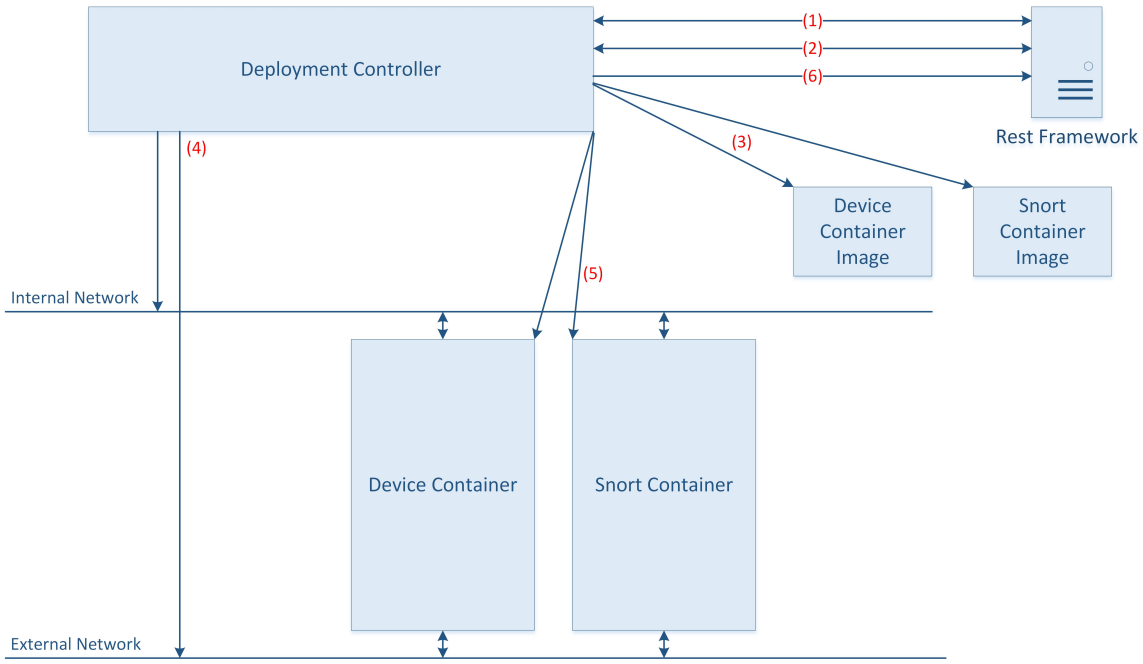

Figure 4.5: Example of a Deployment's Timeline

Once a capture interval has elapsed, and the capture file is closed, the captured traffic is analyzed through Snort's retroactive intrusion detection mode. The Snort rules used in this proof of concept were the default Snort ruleset, which does require an account. The rules contain several entries corresponding to CPS specific exploits, but also are capable of identifying a large amount of other intrusion activity. Once finished, this intrusion detection analysis outputs a list of alerts, as well as a smaller packet capture containing only the traffic that generated those alerts.

In order to test the successful network capture and subsequent identification of exploits included in the configured Snort rules, a Metasploit [43] deployment was used to generate an attack to stop a PLC CPU. The offending packets were correctly identified in the snort logs, as seen in Figure 4.6 and a smaller packet capture was created containing only the offending packets 4.7. Note that the times (as recorded in UTC) agree between logs and the packet capture, and that a full packet capture containing the context for the offending packet is also stored.

Figure 4.6: Exploit Identified by Snort Logs

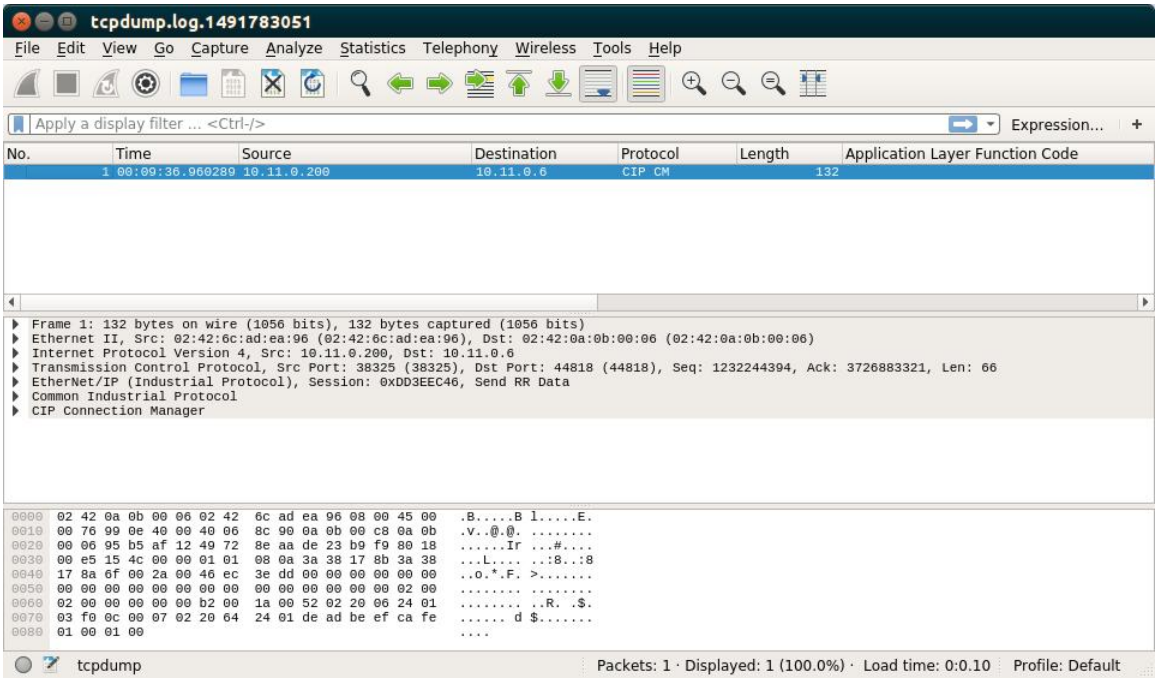

Figure 4.7: Concise Exploit Packet Capture Displayed in Wireshark

Specification for Configuration Files

Multiple configuration files were required to implement the proof of concept. All configuration information is stored as JSON files, while all device specific functionality was implemented as python modules.

The first configuration file, which stores information related to the system being modeled, is called the 'template configuration'. This specified the IP space for the internal network, and enumerates the devices that make up the modeled system. For each device, some meta-data is included, an internal interface and the path to device service modules are specified, and several lists are included. These lists consist of the list of services that device presents, the system variables that device uses, and what other device models that device communicates with.

The second configuration file separates the deployment specific information from the template configuration. This is a much shorter file, and only specifies the IP space of the external network and any external interfaces individual device models might present.

Device service modules are stored in separate directories, and for local deployments device logs are stored in these directories as well.

4.2.2 Framework Elements Specific to the Implemented System Model

The following pieces of the framework deal with implementing the specific system model for this proof of concept.

Modeled System

In order to create an implementation of the framework, it was necessary to select and model a simple system. There exist relatively simple CPS, but system owners are understandably wary of sharing information. To implement the proof of concept then, a simplified version of the water treatment subsystem model used in [21] was reimplemented. The system is elaborated upon in the tutorial offered for their referenced capture the flag [44], and source

code for the example is open sourced at [45]. The source code for the example was analyzed to obtain models for the process and device logic, but no code was reused.

The system architecture can be seen in Figure 4.8. The portion that was modeled consists of tank storage of input water, tank storage of water for ultra-filtration, and the valves and pumps that regulate the different levels in the tanks. In Figure 4.8, input water enters from the left, and the inlet to the Raw Water Storage tank is controlled by Valve 1 and monitored by Flow Meter 1. The water in the storage tank is then pumped through Pump 1, monitored by Flow Meter 2, into the Ultra-Filtration Tank, depending on the tank levels.

The physical devices are monitored and controlled by 3 PLCs, labeled in Figure 4.8 as PLC 1, PLC 2, and PLC 3. PLC 1 is responsible for controlling Valve 1 and Pump 1, measuring Flow Meter 1, and measuring the level of the Raw Water Storage tank. PLC 2 is responsible for measuring Flow Meter 2. PLC 3 is responsible for measuring the level of the Ultra-Filtration tank.

All PLCs communicate as necessary over the common CPS protocol Ethernet/IP. Specifically, PLC 1 executes logic dependent on the flow observed by Flow Meter 2 and the level of the Ultra-Filtration tank, and so queries PLC 2 and PLC 3 for their measured values.

There is additionally a simple HMI which displays the measured values for the Raw Water Storage tank level, as well as the activation status of Valve 1 and Pump 1. This HMI obtains these values by querying PLC 1 over Ethernet/IP. Figures 4.9 and 4.10 show examples of this HMI.

Process Implementation

The first native python implementation of the process was done using an extension of a generic device. A generic process service was created, such that the only methods that a user needs to implement is an `update state` method, which should update process state based on a given update rate, and a `deal with violations` method, which defines how a process should deal with cases where physical boundaries are breached. For

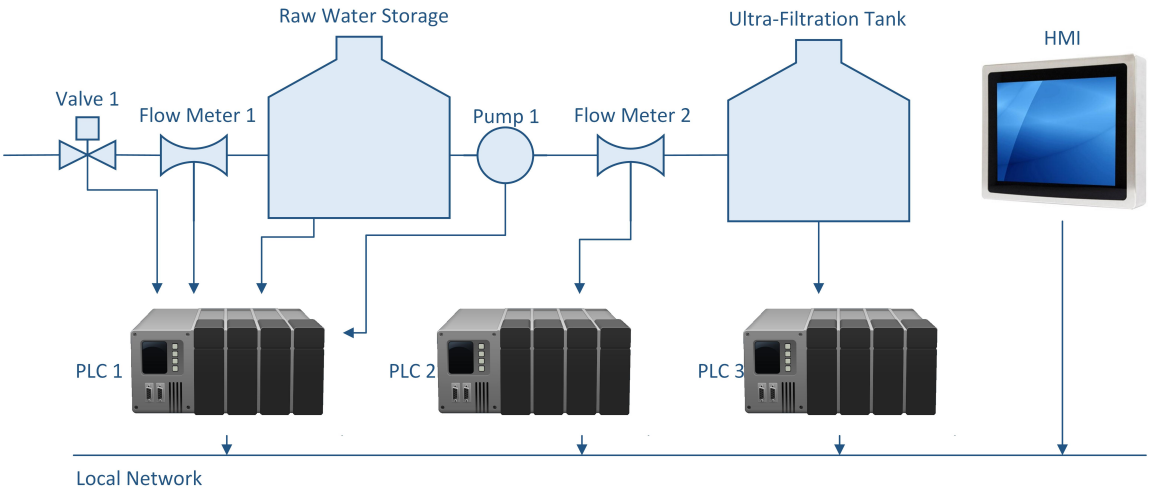

Figure 4.8: Architecture of the Modeled System

this specific system, the process model was reimplemented from the example system, and consisted of updating tank levels by calculating the volume change of fluid in the tanks based on flow rates into and out of the tanks. An example of process behavior as seen through the implemented HMI is seen in Figure 4.9. In this particular figure, the level of the raw storage tank is seen to exceed a threshold, and this triggers PLC1 to close the input valve. Consequently, water continues to drain from the raw storage tank into the ultra-filtration tank, but is not replaced by the input flow.

To illustrate the degree to which framework support eased the reimplementation of the process model, the full native python implementation is included in Appendix A.

External Implementation

In order to prove the feasibility of substituting external process models, a separate process model was implemented using Matlab. This external process model used a naive communication scheme over local TCP with a simple service, which translated between the Matlab communications and the internal interface communications. The same pattern would allow the use of more common pieces of modeling software, such as LabView [46]. The external process model exhibited similar behavior to Figure 4.9, as seen in Figure 4.10. In this

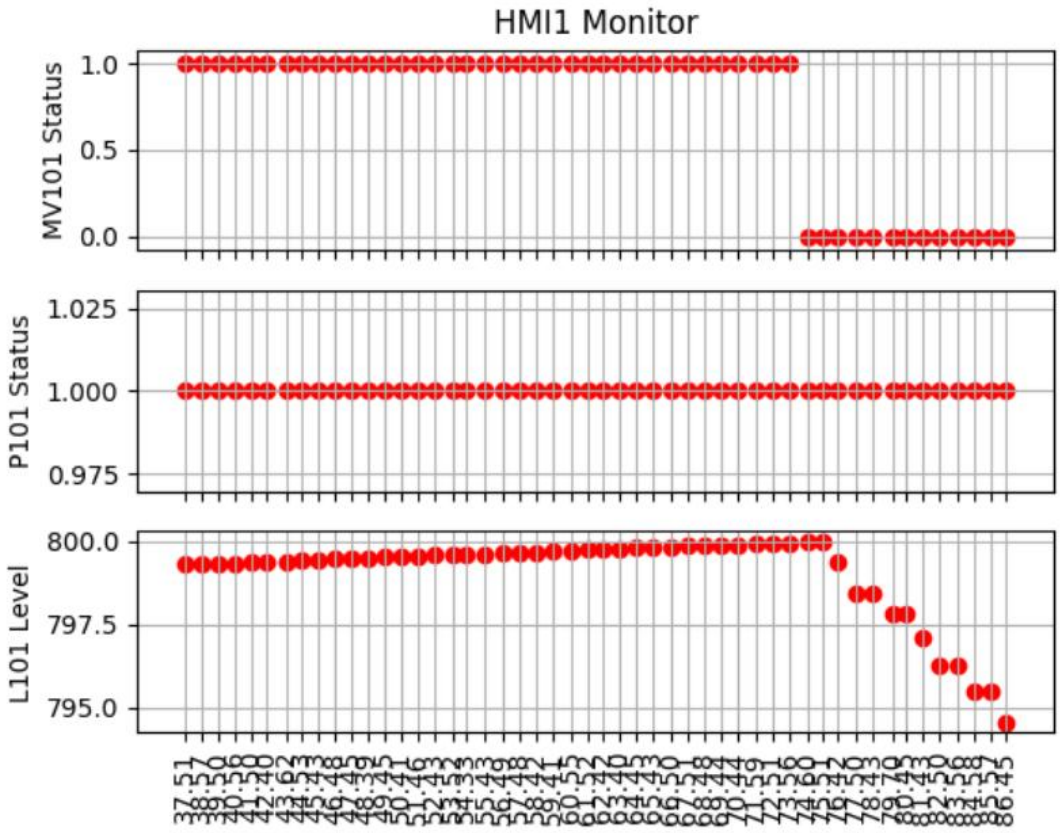

Figure 4.9: HMI View of Native Python Process Model

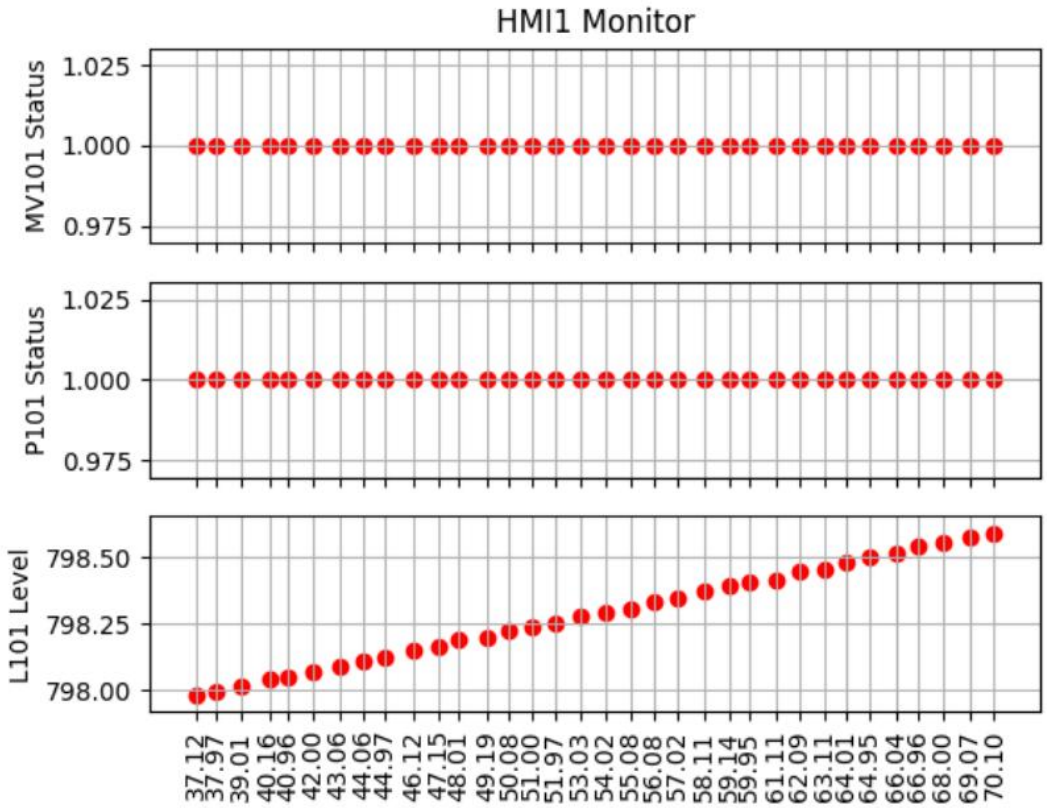

Figure 4.10: HMI View of External Matlab Process Model

particular figure, the process is seen at an early stage, where all valves are open and pumps are on, and the raw storage tank is seen slowly filling. Of note is the fact that no implementation details other than the process model changed, aside from tweaking the number of measurements displayed.

Device Logic Implementations

PLC 1 was required to make decisions based on observed process parameters. Specifically, it opened/closed Valve 1 based on the level of water in the Raw Water Storage tank, and turned on/off Pump 1 based on the level of water in the Ultra-Filtration tank. This was implemented as a service which requested the process parameters, implemented the logic, and issued actuations as necessary. The flow read by Flow Meter 2 and the level of the Ultra-

Filtration tank were read by PLC 2 and PLC 3 respectively, so the Ethernet/IP Services presented by PLCs 1, 2, and 3 were used to enable PLC 1 to read those process variables.

Ethernet/IP Service

All PLCs needed to create and respond to Ethernet/IP traffic. In order to support this, a simple service was implemented to interface with the Python Ethernet/IP server module, CPPPO. Similarly, the HMI was required to read values from PLC 1's Ethernet/IP server, in order to present them to the user, and leverage the same module.

The models implemented for all PLCs are generic, and have no specific target devices. However, there exist several published attacks for common PLC vendors. These exploits are included in the default Metasploit [43] install, and a single simple exploit which stops the PLC CPU was emulated in PLC 3. For this specific exploit, the only observable effect to an external attacker would be cessation of all Ethernet/IP traffic. The exploit was successfully emulated, and when exercised by Metasploit, resulted in a loss of communication both between PLC 3 and external traffic and PLC 3 and other PLCs.

Naive Device Model

In order to implement the actuation devices of Valve 1 and Pump 1, a simple naive device model was first implemented. In this model, when an actuation device receives an actuation from PLC 1, the process model is immediately updated. This is the simplest realizable model, and can be used as a placeholder until more realistic device models are generated, either through direct observation of the physical device or though white box modeling. This model was used for both the native python process implementation and for the external Matlab process implementation, shown in Figures 4.9 and 4.10 respectively.

Delaying Device Model

In implementing this system, no device models were available. In order to prove the feasibility of leveraging such a model though, a device model for Valve 1 was created that introduced a delay to account for the time required for the valve to actuate. In [27], a device model for a relay was developed, and showed that the actuation time distribution was roughly normal. Accordingly, this delay was normally distributed with a mean of 4 seconds and a standard deviation of 0.25 seconds. The effect of this delay can be seen in the highlighted sections of Figure 4.11. This screenshot was taken shortly after the Raw Storage tank level (L101 Level in the figure) exceeded the threshold for closing Valve 1 (MV101 in the figure). The green highlighted time slice shows when the tank level exceeds the threshold (799 mm), and the desired status drops to closed shortly after. The blue highlighted time slice shows when the tank level actually begins dropping, which occurs roughly 4 seconds after the threshold is exceeded, as seen by the timestamps below.

HMI Service

The HMI model implemented in the system was required to serve a website presenting a simple visualization of process behavior. In the modeled system, this visualization was an image which refreshed once a second. In order to accomplish this, two additional services were implemented. One service was supplied with the relevant data from the Ethernet/IP service used in the HMI, and implemented the same data graphing process as the the modeled system, where the most recent set of measurements is displayed for the Raw Water Storage level, Valve 1 on/off status, and Pump 1 on/off status. The other service presented a simple HTTP server, implemented in the Flask Python library. Figures 4.9, 4.10, and 4.11 show examples of the visualization.

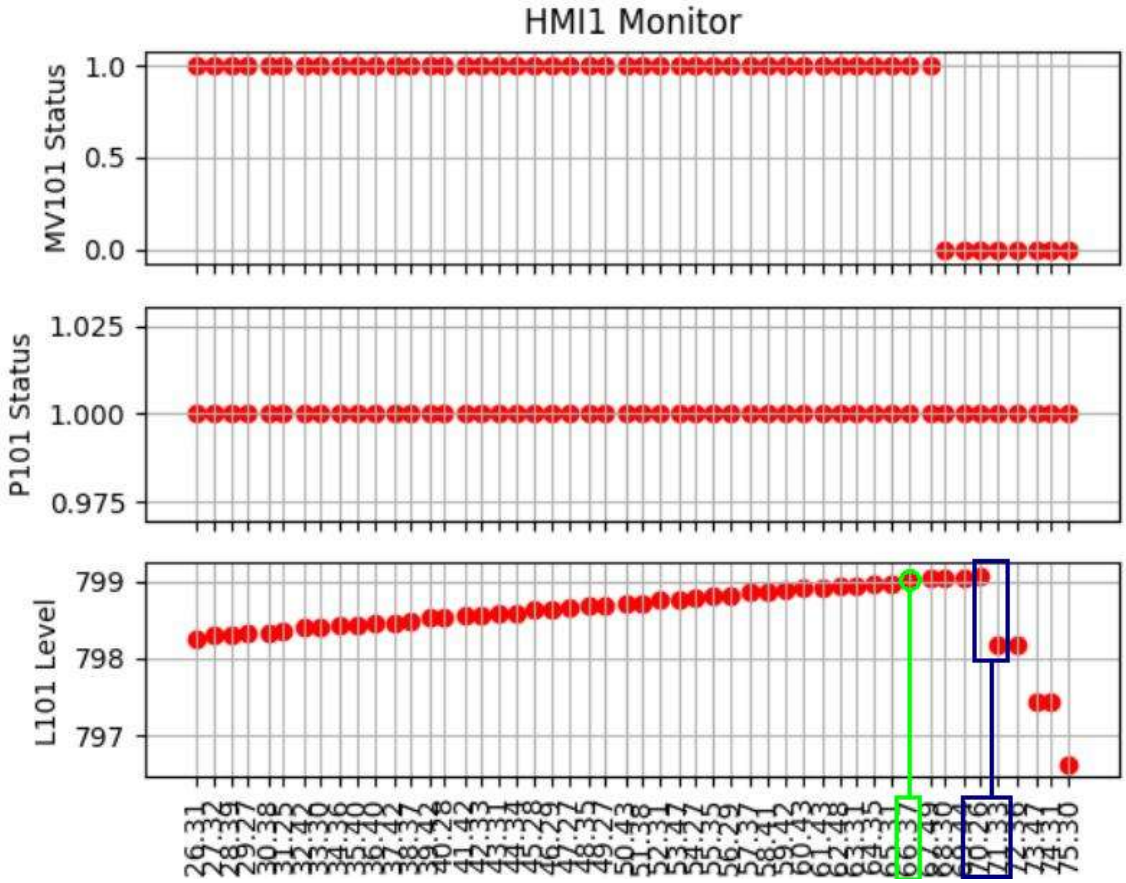

Figure 4.11: Example of Actuation Delay Introduced by Device Model

4.3 Limitations

The proof of concept implementation presented here succeeds in creating a framework capable of simulating a complete system, but still has several recognized limitations.

4.3.1 Simulation Speed

The proof of concept developed here modeled a somewhat slow process, but for systems with faster control loops and faster process evolution, faster implementations will have to be developed. Specifically, the use of the interpreted language Python allowed for ease of development, but is slower than other compiled languages which could potentially be used to model very fast, complicated systems. For the deployable system proof of concept as well, the CPPPO library proved to be a workable, but very slow means of exposing an Ethernet/IP service, and other methods of providing that service would be required for faster systems.

4.3.2 Exploit Emulation

In this thesis, the proof of concept framework was able to emulate a simple DOS exploit for a PLC. However, there exist other, more nuanced exploits of CPS computing devices, and some devices are capable of being reprogrammed by attackers due to poor authentication. An example of leveraging nuanced exploits and PLC reprogramming to implement PLC ransomware is presented in [47]. Emulating these kinds of exploits and reprogramming events is likely not possible using only virtualized devices in the current framework, without implementing a more abstract representation for internal device logic and behavior. Potential solutions include utilizing real devices or locally interpreting the programs run by PLCs, and are discussed in the Future Work section of Chapter 5.

CHAPTER 5
CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this thesis, the design of a new kind of honeypot for CPS is presented.

An initial design is proposed, detailing how various process and device models should interact. Subsequently, a refined design is proposed, detailing a means for easily integrating new models, easily deploying honeypot instances, and centrally collecting logs from a number of honeypot instances.

Two proof of concepts for these designs are implemented. The process of developing process and device models are documented, and the resulting system simulations are presented. The limitations of the refined proof of concept are identified.

5.2 Future Work

While for this thesis, the goal of producing a proof of concept for a deployable CPS honeypot within the HoneyPhy framework was achieved, there exists several promising avenues that would speed and ease adoption in both research and production environments.

5.2.1 Creating High-Interaction Computing Device Models

Some device models developed for this proof of concept exposed services to the external network. Any exposed network service will necessarily need to reflect the network fingerprint of the device it aims to model, and respond to all communication in a realistic way. This can be accomplished through careful emulation, but the simplest means to accomplish this is to connect the actual device. This would, in effect, create a high-interaction device model, which perfectly reflects the device's fingerprint and responses.

In order to integrate this with the current framework, a service will need to be created that interprets signals emitted from the portion of the device that interacts with the physical process, sends actuation events accordingly, and responds with realistic sensor data. The network interface of the device that is exposed to the Internet will also need to be connected to the external network, to allow for potential network communication to other devices.

Furthermore, and this is the difficulty many high-interaction honeypots face, some mechanism for observing and logging device level events must be implemented. Without this, the device becomes a black-box with respect to the rest of the honeypot. This can still be useful to check for attackers capable of causing damage, but does not reveal as much information about attacker methods.

5.2.2 Emulating Reprogramming Events in Low-Interaction Computing Device Models

If high-interaction device models still prove impractical, it is desirable to be still able to emulate nuanced exploits and reprogramming events. A common case of impractical device models will likely be PLCs, where real-time operating systems render virtualization difficult. PLCs do, however, implement their logic using a regular programming language, which is rendered into a binary, and run on a very regular control loop, called a scan cycle.

The possibility of emulating the underlying scan cycle architecture and having device models locally interpret PLC binaries might provide the means to create low-interaction device models capable of emulating reprogramming events.

5.2.3 Automated Intelligence Analysis

With the centrally collected network, process, and device logs provided by a convincing CPS honeypot, the capability and intent of attackers can be assessed in a fine-grained manner. This includes labeling traditional cyber capability, but also an attacker's capability to interact with a process and move it into a dangerous state. Given this new information, attacker interactions can broadly be classified in the following ways:

- Low level reconnaissance: Attacker conducts reconnaissance actions, which betray unauthorized network access, but do not indicate the attacker is in any way aware that the system is related to critical infrastructure. This behavior can come from a variety of sources, depending on external network access.

- Misattributed exploits: Attacker runs exploits, but those exploits are not intended for the modeled devices. This indicates the attacker is unaware of the nature of the modeled system, but has the capability to exploit whatever devices the vulnerability effects.

- General CPS Capability: Attacker conducts reconnaissance actions that indicate familiarity with CPS systems in general, but not the modeled system specifically. This can consist of reading Modbus Tags, Ethernet/IP Device ID information, etc.

- Specific System Capability, but no current malicious intent: The attacker interacts with the system in a way that indicates familiarity with the specific system, but does not indicate any intent to cause harm in any way. This can consist of the first action taken by an unknown host being modifying a state variable, set point, or device status. This indicates severe compromise.

- Specific System Capability and Malicious Intent: The attacker interacts with the system in a way that, were the system real, could cause damage to the system or to the system's operators. This indicates extreme attacker capability as well as malicious intent.

In the future, an automated system could be created to process all collected logs and label interactions in this way. This could be useful in a production environment if the resulting labels are checked periodically by system users or security experts, or if the automated assessments send alerts to an organization's SOC. This will also ease the manual forensics of intrusions in a research context.

5.2.4 Real-time Switching for Intrusion Prevention

There exist some systems, such as [48, 49], which are capable of real time dynamic simulation of complicated CPS. It could be feasible to have these systems integrated into the HoneyPhy framework, and provide their traditional value, but also have the opportunity to identify malicious behaviors and switch to entirely simulated device models.

Such a system would provide the traditional benefits of a production honeypot, but also include convincing enough device models to capture more sophisticated attacker activities for research purposes. In addition, the utilization of the default process model to provide additional and ancillary value to actual system operations could ease adoption among some sectors.

This would require tight coupling with the system owners, as there is always the potential to misidentify legitimate commands as malicious, and block normal operations. Ultimately the feasibility of utilizing a honeypot in-line with an intrusion prevention system should be explored.

5.2.5 Reimplementing To Optimize for Speed

As mentioned above, in order to model systems with fast control loops and high-frequency processes, faster models and services must be developed. This would primarily involve reimplementing the Device Controller and individual services. As the Deployment Controller and the REST Framework have no effect on the speed of individual devices, there would be no need to reimplement them. A compiled language such as c could be used to provide an immediate increase in speed compared to Python. Additionally, a faster method of interprocess communication could be investigated to replace the Multiprocessing Pipes used, as well as the internal interface network.

5.2.6 Full Scale Deployment

While the proof of concepts illustrated the feasibility of creating a convincing CPS honeypot, a full scale deployment would further prove the usefulness of the framework. This would require significant industry collaboration, as device and process models would need to be developed for the specific system to be modeled. Furthermore, industry IP addresses would be ideally used to provide a realistic network presence.

Appendices

APPENDIX A

NATIVE PYTHON PROCESS MODEL

```
#!/usr/bin/python2.7

import math
import src.interfaces.process_interface as process_interface

# Define some constants
TANK_DIAMETER = 1.38  # In m
TANK_RADIUS = TANK_DIAMETER/2  # In m
PUMP_FLOWRATE_IN = 2.55  # In m^3/h
PUMP_FLOWRATE_OUT = 2.45 # In m^3/h

# Define tank set points in mm
l101_constraints = {
    'LL': 250.0,
    'L':  500.0,
    'H': 800.0,
    'HH': 1200.0
}

l301_constraints = {
    'LL': 250.0,
    'L': 800.0,
    'H': 1000.0,
    'HH': 1200.0
}

TANK_HEIGHT = 1.6 # In m
```

```
28
29
30 class Service(process_interface.ProcessInterface):
31
32     def __init__(self, **kwargs):
33
34         # specify device_config.json vars, and assign 'zero' vals to
               them
35
36         self.log_name = 'service_swat_process'
37
38         self.internal_vals['l101'] = 0.0
39         self.internal_vals['l301'] = 0.0
40         self.internal_vals['f101'] = 0.0
41         self.internal_vals['f201'] = 0.0
42         self.internal_vals['p101_real'] = False
43         self.internal_vals['mv101_real'] = False
44
45         # Define the update frequency
46         self.update_period = .2  # 200 ms
47
48         # Define ancillary conversion values
49         self.lvl2vol = math.pi * math.pow(TANK_RADIUS, 2)
50         self.vol2lvl = 1.0 / (math.pi * math.pow(TANK_RADIUS, 2))
51         self.flow2vol = self.update_period / 3600.0
52
53         super(Service, self).__init__()
54
55     def update_state(self):
56
57         # Update flowrates based on valve status
58         if self.internal_vals['mv101_real']:
59             self.internal_vals['f101'] = PUMP_FLOWRATE_IN
```

```
60            else:
61                self.internal_vals['f101'] = 0.0
62
63            if self.internal_vals['p101_real']:
64                self.internal_vals['f201'] = PUMP_FLOWRATE_OUT
65            else:
66                self.internal_vals['f201'] = 0.0
67
68            # Update the tank levels based on their inputs
69
70            # Get current volumes
71            cur_lvl_101 = self.internal_vals['l101']  # l101 contains tank
                  level in mm
72            cur_lvl_101 = cur_lvl_101 / 1000.0 # get level in m
73
74            cur_vol_101 = cur_lvl_101 * self.lvl2vol
75
76            cur_lvl_301 = self.internal_vals['l301']
77            cur_lvl_301 = cur_lvl_301 / 1000.0
78
79            cur_vol_301 = cur_lvl_301 * self.lvl2vol
80
81            # Account for inflow rate
82            cur_vol_101 += self.internal_vals['f101'] * self.flow2vol
83            cur_vol_301 += self.internal_vals['f201'] * self.flow2vol
84
85            # Account for outflow rate
86            cur_vol_101 -= self.internal_vals['f201'] * self.flow2vol
87            # Tank 301 doesn't currently have outputs
88
89            # Write to kv-store, and reconvert to mm
90            self.internal_vals['l101'] = cur_vol_101 * self.vol2lvl * 1000
91            self.internal_vals['l301'] = cur_vol_301 * self.vol2lvl * 1000
```

```
92
93      def deal_with_violations(self, violations_list):
94          pass
95
96      def start_service(self, interface_dict=None, pipe=None, varlist=None
            ):
97
98          super(Service, self).start_service(interface_dict, pipe, varlist
                )
99
100         self.process_loop()
101
102
103 def main():
104
105     swat_process = Service()
106     swat_process.start_service()
107
108
109 if __name__ == "__main__":
110     main()
```

REFERENCES

[1] NIST CPS Public Working Group. (2017). Cyber-physical systems - nist, [Online]. Available: https://www.nist.gove/el/cyber-physical-systems.

[2] NERC. (2017). Standard tpl-001-1 system performance under normal conditions, [Online]. Available: http://www.nerc.com/files/TPL-001-1.pdf.

[3] R. Langner, "To kill a centrifuge," Tech. Rep., 2013. [Online]. Available: https://scadahacker.com/library/Documents/Cyber_Events/Langner%20-%20To%20Kill%20a%20Centrifuge.pdf.

[4] B. Bencsth, G. Pk, L. Buttyn, and M. Flegyhzi, "Duqu: A stuxnet-like malware found in the wild," Tech. Rep., 2011. [Online]. Available: https://www.crysys.hu/publications/files/bencsathPBF11duqu.pdf.

[5] K. Lab, "The duqu 2.0: Technical details," Tech. Rep., 2015. [Online]. Available: https://scadahacker.com/library/Documents/Cyber_Events/Kaspersky%20-%20The%20Duqu%202%20-%20Technical%20Details%20v2.1.pdf.

[6] S. S. Response, "Dragonfly: Cyberespionage attacks against energy suppliers," Tech. Rep., 2014. [Online]. Available: https://scadahacker.com/library/Documents/Cyber_Events/Symantec%20-%20Security%20Response%20-%20Dragonfly%20v1.2.pdf.

[7] R. M. Lee, M. J. Assante, and T. Conway, "Analysis of the cyber attack on the ukranian power grid," Tech. Rep., 2016. [Online]. Available: https://ics.sans.org/media/E-ISAC_SANS_Ukraine_DUC_5.pdf.

[8] J. Matherly. (2017). Shodan, [Online]. Available: https://shodan.io.

[9] C. Fachkaha, E. Bou-Harb, A. Keliris, N. Memon, and M. Ahamad, "Internet-scale probing of CPS: Inference, characterization and orchestration analysis," in *Proceedings of NDSS '17*, 2017.

[10] B. Cheswick, "An evening with berferd in which a cracker is lured, endured, and studied," in *Proc. Winter USENIX Conference, San Francisco*, 1992, pp. 20–24.

[11] N. Provos. (2008). Developments of the honeyd virtual honeypot, [Online]. Available: http://honeyd.org/.

[12] A. Morris. (2014). Detecting kippo ssh honeypots, bypassing patches, and all that jazz, [Online]. Available: https://morris.sc/detecting-kippo-ssh-honeypots/ (visited on 04/10/2017).

[13] (2017). Sebek, [Online]. Available: https://projects.honeynet.org/sebek/.

[14] M. Dornseif, T. Holz, and C. N. Klein, "Nosebreak - attacking honeynets," in *Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, 2004.*, 2004, pp. 123–129.

[15] V. Pothamsetty and M. Franz. (2005). Scada honeynet project: Building honeypots for industrial networks, [Online]. Available: http://scadahoneynet.sourceforge.net/.

[16] Digital Bond. (2016). Scada honeynet, [Online]. Available: http://digitalbond.com/tools/scada-honeynet/.

[17] The Honeynet Project. (2016). Honeywall, [Online]. Available: https://projects.honeynet.org/honeywall/.

[18] T. Vollmer and M. Manic, "Cyber-physical system security with deceptive virtual hosts for industrial control networks," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 2, pp. 1337–1347, May 2014.

[19] K. Wilhoit and S. Hilt, *The gaspot experiment: Unexamined perils in using gas-tank-monitoring systems*, 2015. [Online]. Available: https://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/whitepapers/wp_the_gaspot_experiment.pdf.

[20] M. Winn, M. Rice, S. Dunlap, J. Lopez, and B. Mullins, "Constructing cost-effective and targetable industrial control system honeypots for production networks," *International Journal of Critical Infrastructure Protection*, vol. 10, pp. 47 –58, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1874548215000256.

[21] D. Antonioli, A. Agrawal, and N. O. Tippenhauer, "Towards high-interaction virtual ics honeypots-in-a-box," in *Proceedings of the 2Nd ACM Workshop on Cyber-Physical Systems Security and Privacy*, ser. CPS-SPC '16, Vienna, Austria: ACM, 2016, pp. 13–22, ISBN: 978-1-4503-4568-2. [Online]. Available: http://doi.acm.org/10.1145/2994487.2994493.

[22] D. Antonioli and N. O. Tippenhauer, "Minicps: A toolkit for security research on CPS networks," *CoRR*, vol. abs/1507.04860, 2015. [Online]. Available: http://arxiv.org/abs/1507.04860.

[23] L. Rist, J. Vestergaard, D. Haslinger, and A. Pasquale. (2016). Conpot, [Online]. Available: http://conpot.org.

[24] C. Scott, "White paper: Designing and implementing a honeypot for a scada network," SANS Institute Infosec Reading Room, Tech. Rep., 2014.

[25] GridPot. (2017). Gridpot github project, [Online]. Available: https://github.com/sk4ld/gridpot.

[26] D. Buza, F. Juhasz, M. Felegyhazi, and T. Holczer, "Cryplh: Protecting smart energy systems from targeted attacks with a plc honeypot," in *Smart Grid Security: Second International Workshop*, 2014, pp. 181–192.

[27] D. Formby, P. Srinivasan, A. Leonard, J. Rogers, and R. Beyah, "Who's in control of your control system? device fingerprinting for cyber-physical systems," *NDSS*, Feb. 2016.

[28] Y. Shoukry, P. Martin, Y. Yona, S. Diggavi, and M Srivastava, "Py-cra: Physical challenge-response authentication for active sensors under spoofing attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2015, pp. 1004–1015.

[29] D. Haziosmanovic, R. Sommer, E. Zambon, and P. Hartel, "Through the eye of the plc: Semantic security monitoring for industrial processes," in *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014, pp. 126–135.

[30] Threatstream. (2017). Modern honey network, [Online]. Available: https://github.com/threatstream/mhn.

[31] (2017). Hpfeeds, [Online]. Available: https://github.com/rep/hpfeeds.

[32] V. Aliyev, "Using honeypots to study skill level of attackers based on the exploited vulnerabilities in the network," 2010.

[33] (2017). Ics-cert, [Online]. Available: https://ics-cert.us-cert.gov/.

[34] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling, "The nepenthes platform: An efficient approach to collect malware," in *Recent Advances in Intrusion Detection: 9th International Symposium, RAID 2006 Hamburg, Germany, September 20-22, 2006 Proceedings*, D. Zamboni and C. Kruegel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 165–184, ISBN: 978-3-540-39725-0. [Online]. Available: http://dx.doi.org/10.1007/11856214_9.

[35] (2017). Dionaea - catches bugs, [Online]. Available: https://github.com/DinoTools/dionaea.

[36] J. Guarnizo, A. Tambe, S. S. Bhunia, M. Ochoa, N. O. Tippenhauer, A. Shabtai, and Y. Elovici, "SIPHON: towards scalable high-interaction physical honeypots," *CoRR*, vol. abs/1701.02446, 2017. [Online]. Available: http://arxiv.org/abs/1701.02446.

[37] (2017). Django: The web framework for perfectionists with deadlines, [Online]. Available: https://www.djangoproject.com/.

[38] (2017). Django rest framework, [Online]. Available: http://www.django-rest-framework.org/.

[39] Docker. (2017). Docker, [Online]. Available: https://www.docker.com/.

[40] (2017). A python library for the docker engine api, [Online]. Available: https://github.com/docker/docker-py.

[41] (2017). Snort - network intrusion detection & prevention system, [Online]. Available: https://www.snort.org/.

[42] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2015, pp. 171–172.

[43] (2017). Penetration testing software — metasploit, [Online]. Available: https://www.metasploit.com.

[44] (2017). Swat tutorial – minicps 1.0.0 documentation, [Online]. Available: http://minicps.readthedocs.io/en/latest/swat-tutorial.html#system-overview.

[45] (2017). Minicps github repository, [Online]. Available: https://github.com/scy-phy/minicps.

[46] (2017). Labview system design software, [Online]. Available: http://www.ni.com/labview/.

[47] D. Formby, S. Durbha, and R. Beyah. (2017). Out of control: Ransomware for industrial control systems, [Online]. Available: http://www.cap.gatech.edu/plcransomware.pdf.

[48] S. Meliopoulos, R. Huang, E. Polymeneas, and G. Cokkinides, "Distributed dynamic state estimation: Fundamental building block for the smart grid," in *2015 IEEE Power Energy Society General Meeting*, 2015, pp. 1–6.

[49] S. Choi, B. Kim, G. J. Cokkinides, and A. P. S. Meliopoulos, "Feasibility study: Autonomous state estimation in distribution systems," *IEEE Transactions on Power Systems*, vol. 26, no. 4, pp. 2109–2117, 2011.